Jül-4344

Jülich Supercomputing Centre (JSC)

A Web Framework for Workflow Submission and Monitoring via UNICORE 6 based on Distributable Scientific Workflow Templates

S. Bergmann

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

A Web Framework for Workflow Submission and Monitoring via UNICORE 6 based on Distributable Scientific Workflow Templates

S. Bergmann

Berichte des Forschungszentrums Jülich; 4344
ISSN 0944-2952
Jülich Supercomputing Centre (JSC)
Jül-4344

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Abstract

Grid middleware systems like UNICORE allow users to submit Grid jobs and perform computational work. Workflows, modelled as compound directed graphs and described by XML documents, can be created and manipulated by Grid workflow experts in a workflow editor provided by the UNICORE Rich Client. Parameters are used to describe job definitions presented as nodes, and enclosing structures visualised through subgraphs in the workflow structure.
A new export function developed for this Master thesis allows the selection of template parameters and produces a workflow template. Template parametrisation is much easier than workflow creation from scratch.
Furthermore, a Web application based on Apache Wicket has been developed, which provides a repository for workflow templates and functions for workflow processing. Groovy components, generated through the export function, allow end-users a manipulation of workflow templates by specifying new template parameter values.

Zusammenfassung

UNICORE ist ein Grid-Middleware-System, welches Benutzern ermöglicht, Jobs zu submittieren und wissenschaftliche Berechnungen auf Hochleistungsrechnern durchzuführen. Workflows können mithilfe des grafischen UNICORE Rich Client (URC) erstellt und verändert werden. Deren Ausführung geschieht über das UNICORE Workflow System. Workflow-Parameter dienen dazu, Eigenschaften von Jobs und Strukturen, die im Workflow definiert sind, festzulegen. In dieser Arbeit wurde eine neue Export-Funktion für den Workflow-Editor des URC entwickelt, welche es erlaubt, Workflow-Parameter zu selektieren, die durch Platzhalter in der Workflow-Beschreibung ersetzt werden. Die dadurch entstandenen Workflow-Templates ermöglichen eine stark vereinfachte Benutzeroberfläche zur Wiederverwendung von Workflows. Dabei können neue experimentelle Ergebnisse durch die Anpassung von Workflow-Parametern erzielt werden. Hierzu wurde ein Web Framework mithilfe von Apache Wicket implementiert, welches eine einfache Manipulation und Submission von Workflow-Templates erlaubt. Die Ausführung submittierter Workflows kann überwacht werden und Ergebnisdateien können über das Web Framework heruntergeladen werden.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation and problem definition

Grid middleware systems like the Uniform Interface to Computing Resources (UNICORE[1]) manage the coordinated use of distributed computer resources, allowing users to get access to them. UNICORE enables users to perform remote computations by submitting work descriptions, the so-called Grid jobs. One extension of the UNICORE base system is the UNICORE workflow system.
In this system, workflows are modelled with acyclic compound directed graphs. The definition of compound directed graphs is obtained from the technical report *Layout of Compound Directed Graphs* from Georg Sander [16]. A directed graph $G = (V,E)$ consists of a set of nodes V and a set of edges $E \subseteq VxV$. For an edge $(v,w) \in E$ the notation is $v \rightarrow w$ and $v \xrightarrow{*} w$ for a (potentially empty) sequence of edges $v \rightarrow \mathrm{v_1}, \mathrm{v_1} \rightarrow \mathrm{v_2}, ...\mathrm{v_n} \rightarrow w$ (a path). A directed acyclic graph (so-called DAG) contains no cycles defined as a non empty path $v \xrightarrow{*} v$. A tree is a DAG $T = (V,E)$ consisting of n nodes and n-1 edges which has a special root node $r \in V$ with $r \xrightarrow{*} v$ for each $v \in V$. The leaves of the tree have the property that they have no outgoing edges. All other nodes are called inner nodes of the tree. A compound directed graph $C = (G',T')$ consists of a simple directed graph $G' = (B \cup S, \mathrm{E_G})$ and a tree $T' = (B \cup S, \mathrm{E_T})$. The set B contains the leaves of T' which are called base nodes, and the set S contains the inner nodes of T' which are called subgraphs. In a compound graph, $\mathrm{E_G}$ (i.e. the edges of G') represent a connectivity relation between the base nodes and subgraphs. $\mathrm{E_T}$ (i.e. the edges of T') represent a nesting relation: subgraphs may contain other subgraphs or base nodes. A compound graph may contain connectivity edges that cross the borders of nested subgraphs. Thus it is not recursively defined as a graph of subgraphs which can contain subgraphs etc. The base nodes of a UNICORE workflow represent Grid jobs and other activities that are seen as atomic, whereas the subgraphs represent loops and If-Else statements. The edges of a workflow define the execution order. Workflows are described in Extensible Markup Language (XML) documents [17].
UNICORE consists of client and server parts. The UNICORE Rich Client (URC) allows users to create scientific workflows in a graphical workflow editor. In order to use all complex features provided by the URC, the users should have knowledge about workflow creation. Thus these users are called Grid workflow experts.
Loops for example are described by an iteration variable that takes a different value for each loop iteration and an end condition. The iteration variable is a label to distinguish between iterations as well as a replaceable parameter.

[1] UNICORE: http://www.unicore.eu

One of the aims of this work is to find and substitute workflow parameters with placeholders for generating workflow templates. The parameter values, which are to be substituted by placeholders, are called template parameters. Template parameters are distinct concerning their type. Besides primitive parameters (strings, Boolean, integers, floating point numbers), files and file sets exist. A workflow template is characterised by a submittable workflow description containing placeholders for template parameters. It can be filled with new template parameter values to build concrete workflows which can be submitted to the UNICORE workflow system. The automatic workflow template generation takes place in an export function in the URC to be implemented in this Master thesis. Template parameters will be selected during the export process. Workflow templates are useful to build concrete workflows without having knowledge about workflow creation. In research projects end-users require simple access to prefabricated workflow templates to execute workflows with changing parameters for each scenario.
The integration system for workflow templates will be a Web application because it provides world-wide access without installation of specialised client software. The Web application will be designed as a framework that is extensible with new functionality. The application will be based on the open source project Apache Wicket, which facilitates the development of Web applications. The exported templates will be made accessible by uploading them to the Web application. A simple Web form will allow users to change template parameter values and submit the finalised workflow template to the UNICORE workflow system.
Furthermore, the Web application should provide monitoring and output fetching for easy access from everywhere over the network.
As an example, the UIMA-HPC[2] project deals with the generation of workflows, which are used to extract knowledge from unstructured data and make the results accessible in a structured form to the end-user. The workflow structure is always the same, whereas the input data to be analysed change in every scenario. The workflow structure can be created in the URC workflow editor by a Grid workflow expert. This is followed by a workflow template generation step where the input file set is selected as template parameter of the workflow template. Then the end-user can add new files to the selected file set and submit the finalised workflow template via the Web application without creating the workflow from scratch.

1.2 Document structure

This document is structured as follows: Chapter two introduces the fundamental principles of UNICORE workflow processing through the workflow system, Web application framework, especially Apache Wicket, and the scripting language Groovy. The concrete requirement analysis is performed in Chapter three.
The fourth chapter describes the system architecture. Chapter five comprises a design description of the export function and the Web application whose implementation is explained in Chapter six. The whole procedure from template creation to submission to the UNICORE workflow system is verified using an expedient workflow example in Chapter seven. The example shows the process of 3D video rendering using the Blender[3] application. The last chapter concludes the thesis by showing the outcome for researchers and users and by exploring possibilities for further developments.

[2]UIMA-HPC: http://www.fz-juelich.de/SharedDocs/Meldungen/IAS/JSC/EN/2011/2011-04-uima-hpc.html
[3]Blender: http://www.blender.org/

Chapter 2

Prerequisites

This chapter lays the foundation for understanding the work done in this Master thesis. It starts with an introduction into Grid computing and Grid middleware systems, followed by a short analysis of the Grid middleware UNICORE. For the Web application implementation it is necessary to explain the essentials and to analyse the chosen Web application framework technology Apache Wicket. The scripting language Groovy is used for the integration of workflow templates. How it works and why it is used, is described at the end of the chapter.

2.1 Grid computing and Grid middleware systems

The term "Grid computing" was coined in „The Grid: Blueprint for a New Computing Infrastructure" [11] in 1998. Foster and Kesselman state that the Grid allows users to share raw computing power, which enables one user the use of full capabilities. It should be as easy as possible to get access to independent resources which are combined to form a single infrastructure. SETI@home[1] is one successful project where networked PC's world-wide were used to solve compute-intensive research problems. Grid infrastructures are networks of connected computing resources used by communities and projects. The coordination of a Grid infrastructure is realised by a Grid middleware. Such a Grid middleware hides the heterogeneity of the system and enables users to access shared computing resources in a secure and seamless manner. One of these Grid middleware systems is UNICORE which provides Grid services for job submission, monitoring etc. Other Grid middleware systems are for example gLite[2] and Globus[3].

2.2 UNICORE

The first UNICORE project started in 1997. The result in 1999 was the first UNICORE version. In 2004 UNICORE became open source under the BSD license.

[1] SETI@home: http://setiathome.berkeley.edu
[2] gLite: http://glite.cern.ch/
[3] Globus: http://www.globus.org/

Since then, UNICORE has been developed further in different projects like the European Middleware Initiative (EMI[4]) or Networks of Knowledge in the Grid (WisNetGrid[5]), which deal with the development of UNICORE extensions. The EMI project represents a collaboration of major European middleware providers such as gLite and UNICORE. One of its major tasks is to extend the interoperability between Grid infrastructures. WisNetGrid aims at creating a common "knowledge space" within the D-Grid[6] infrastructure for education and research in Germany.
The developer community of UNICORE comprises members from Germany, Poland, Russia, Italy, UK and other countries. The main contributor is the Jülich Supercomputing Centre at Forschungszentrum Jülich GmbH[7]. UNICORE is implemented in the object-oriented programming language Java and thereby platform independent. UNICORE is also extensible by plugins which allow to improve Grid service functionality.
Organisations like the Open Grid Forum (OGF[8]) and the Organization for the Advancement of Structured Information Standards (OASIS[9]) are engaged in the development of Grid standards, which are used to increase the interoperability between Grid infrastructures. UNICORE complies to those standards for job submission and execution monitoring.
Secure access to computing resources is currently implemented by using asymmetric encryption with X509 certificates. User certificates are signed by a Certification Authority (CA). Only certificates signed by trustworthy CA's are allowed for secure access to computing resources. This access control is used to avoid long user lists. The access control is based on XACML policies [4], another OASIS standard, which describes an XML policy language.
In order to communicate with the UNICORE server and all its services, different client interfaces are available. Clients like the UNICORE Rich Client (URC) are user friendly and easy to use by providing graphical user interfaces. The UNICORE Command line Client (UCC) on the other hand, has a very simple command line API (Application Programming Interface) for job submission and monitoring. Portal client solutions with UNICORE support are developed in projects like GridSphere[10] or Vine Toolkit[11]. GridSphere enables developers to engineer and package third-party portlet Web applications, whereas the Vine Toolkit is a modular, extensible Java library including a high-level API for Grid-enabling applications. A UNICORE Web client, which offers functions like the URC and the UCC over the Web, is still work in progress.

2.2.1 UNICORE architecture

Figure 2.1 visualises the core architecture of UNICORE [18]. It is divided in three layers: the client, service and target system layer. The following bottom-up layer description is useful for understanding the architecture.

[4]EMI: http://www.eu-emi.eu/
[5]WisNetGrid: http://www.wisnetgrid.org/
[6]D-Grid: http://www.d-grid-ggmbh.de/
[7]Forschungszentrum Jülich GmbH: http://www.fz-juelich.de
[8]OGF: http://www.gridforum.org/
[9]OASIS: http://www.oasis-open.org/
[10]gridsphere: http://www.gridsphere.org/gridsphere/gridsphere
[11]vinetoolkit: http://vinetoolkit.org/

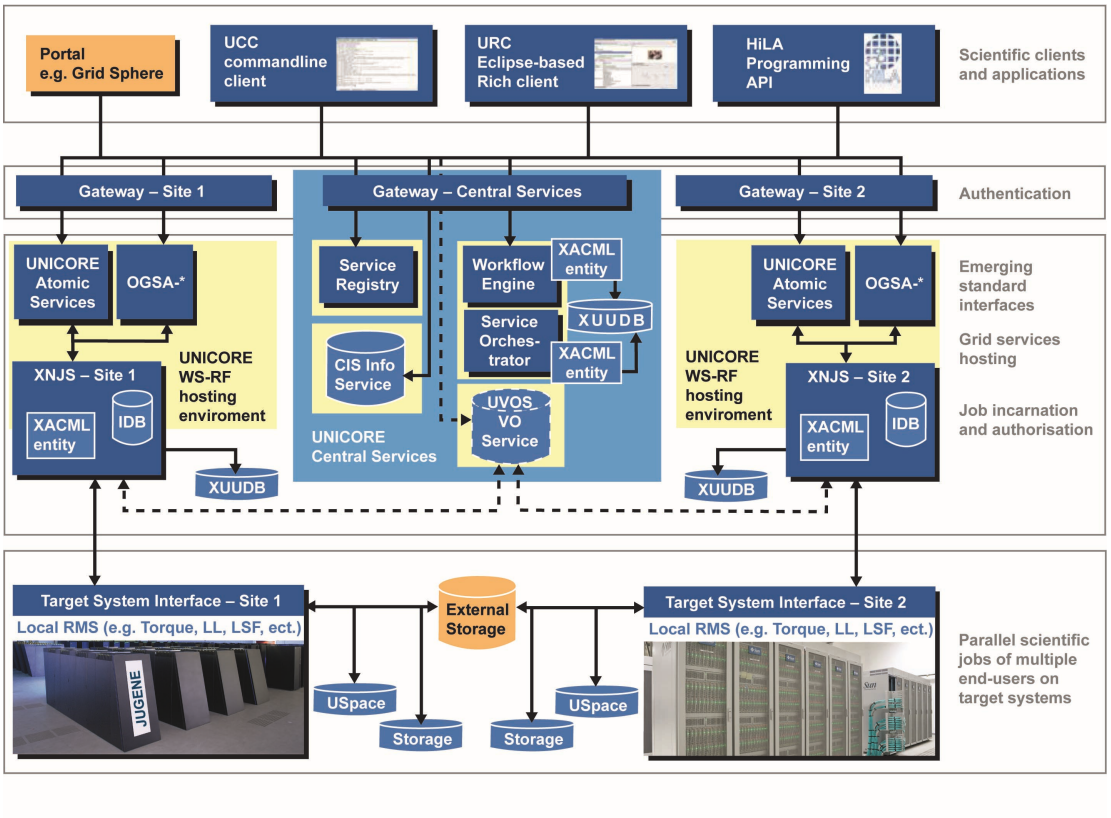


Figure 2.1: UNICORE basis architecture, source [18]

UNICORE target system layer

The target system layer contains a network protocol, the so-called Target System Interface (TSI) for the communication between the UNICORE services and the computing resources. An appropriate TSI implementation is responsible for the execution of system-specific commands.

UNICORE services

The service layer of the UNICORE architecture comprises all available services like gateway, XNJS, XUUDB and registry. The gateway acts as an entry point to a site and performs request authentications. The gateway works like an open door in the firewall. Only requests including trustworthy certificates are allowed. The XNJS is a job processing engine that forwards incoming jobs to the site's batch system (via the target system interface, TSI) and monitors their execution state. It maps the abstract job description to a specific job using rules stored in the Incarnation Data Base (IDB), for example job arguments are converted to system specific paths and variables. It can also be used to transfer files to the site's local file systems (again via the TSI).

In order to get access to an XNJS component, two Web service interfaces are available, the UNICORE Atomic Services (UAS e.g. target system service (TSS), job management service (JMS), storage management service (SMS) and a standardised set of interfaces described in the OGSA-BES specification [10]. The UAS are stateful Web services as defined in the Web service resource framework (WSRF[12]). That means that in contrast to plain Web services (WS-I[13] specification) they have a defined life time and a set of resource properties representing their current state (e.g. execution state of a job management service). The lifetime and all resource properties can be obtained through the so-called resource property document, which is an XML document. The design of the UAS is heavily based on a factory design pattern where one Web service resource type acts as a factory for another Web service resource type. For instance, a target system service creates a new job management service upon job submission. The newly created job management service can then be used for monitoring and steering job execution.
The XUUDB, a database including user certificates, is used for user authentication. Besides those services that are installed on every UNICORE site some central services exist. All available UNICORE services are reported to a registry, which provides a list of all services. Another central service is the Common Information Service [14] (CIS) which delivers detailed information of all connected XNJS components. The UNICORE Virtual Organisation System[14] (UVOS) is used for user authentication similarly to the XUUDB. Since it is based on standards, it is also possible to use UVOS with other Grid middleware systems. Compared to the XUUDB, UVOS allows for fine-grained access control. For workflow processing, the workflow engine and the service orchestrator are needed. They are described in detail in chapter 2.2.2.

UNICORE clients

The High Level API for Grid applications (HiLA [19]) allows the simple integration of UNICORE 6 in user applications. In addition to HiLA the user communication with the service layer can be realised by different UNICORE client systems, which are extensible with plugins. The first system is the UCC [5]. It allows to access all UNICORE service functionalities in a shell or scripting environment. The UCC offers commands to run jobs, monitor their status and fetch their output.
The URC is a graphical client based on the Eclipse Rich Client Platform (RCP), which provides the basis for the export function to be implemented. Figure 2.2 shows a snapshot of the URC and its layout of an RCP workbench window [8]. Two different main components are visible: two views (1 and 2) and one editor (3). The Grid browser (1) is a tree based view including all available bookmarked Registries. Beneath these bookmarks there is a list of all services known to the registry. In the example screenshot, the list of services consists of one workflow engine, three UNICORE execution sites and two UNICORE storage management services. The workflow engine is described in the next chapter. The URC provides a graphical workflow editor (3) for creating and modifying workflows. Figure 2.2 shows the editor for a workflow called *ExampleWorkflow2.flow*, which contains one script job and one If-Else structure with tasks defined for the if and else branch. Control flows, visualised by arrows, are used to specify the order of job processing.

[12]WSRF: http://www.oasis-open.org/committees/wsrf/
[13]WS-I: http://www.ws-i.org/
[14]UVOS: http://uvos.chemomentum.org/

In the presented example, *Script1* is the first job which will be processed. The exit code of *Script1* decides the following action: If *Script1* was executed successfully (resulting in an exit code of 0), *Job1* will be processed, otherwise *Script2*. Besides sequential processing it is also possible to define workflows where jobs will be executed in parallel.
A palette including different actions is shown on the left hand side of the workflow editor. The user can select among different applications, defined in graphical interfaces, the so-called GridBeans: the Script GridBean, the Blender GridBean, the Povray GridBean and the Generic GridBean. The Script GridBean provides a list of script applications like bash or Perl whereas the Povray and the Blender GridBean offer an interface for POV-Ray[15] and Blender specific arguments.

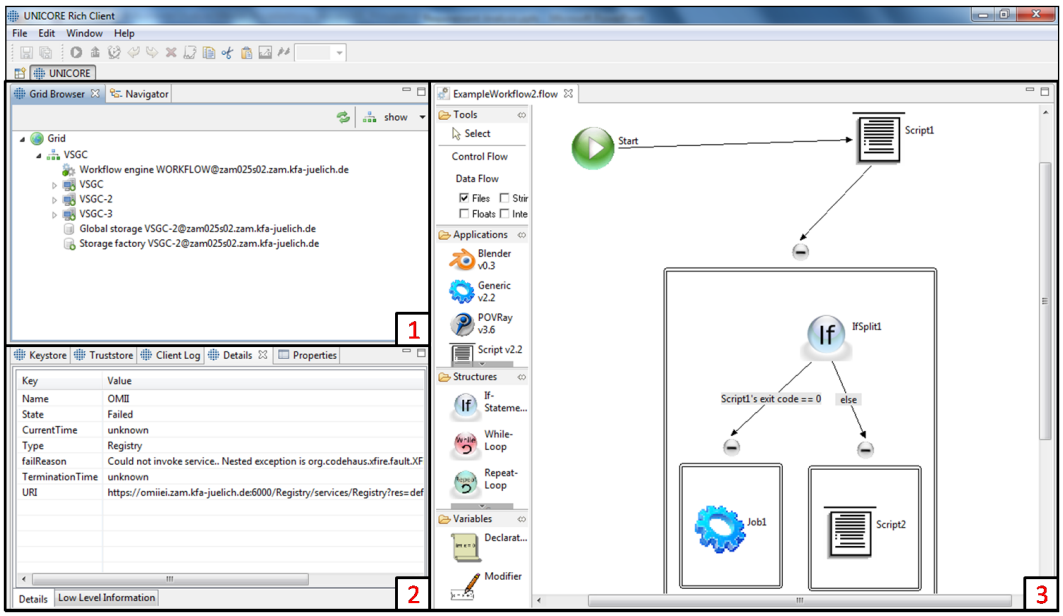


Figure 2.2: UNICORE Rich Client

The Generic GridBean is used to define a job that executes an arbitrary application installed on a target system with specified arguments. Next to If-Else and Group structures it is also possible to create While, For-Each and Repeat loops, each of which consists of iteration variables, variable modifiers and terminal conditions. The iteration variables are used to differentiate between iteration numbers whereas variable modifiers change the iteration variable value for each iteration. A loop performs integrated tasks until the value of the iteration variable is equivalent to the terminal condition. Further palette entries are useful to declare or modify workflow variables.
After workflow creation, the users can submit the workflow to a workflow engine of their choice.

[15]POV-Ray: http://www.povray.org/

2.2.2 UNICORE workflow system

The workflow system is used for workflow processing, execution and monitoring [3]. The two main components, workflow engine and service orchestrator, are responsible for workflow execution.
Figure 2.3 presents a simplified sequence diagram for the processing of a small workflow including one Grid job. After workflow creation the client submits the workflow to the workflow engine. Analogously to the target system service the workflow engine creates a workflow management service instance when a client submits a workflow. Similar to the job management service, the workflow management service can be used for monitoring and steering. Both workflow engine and workflow management service are stateful Web services compliant to the WSRF specification.

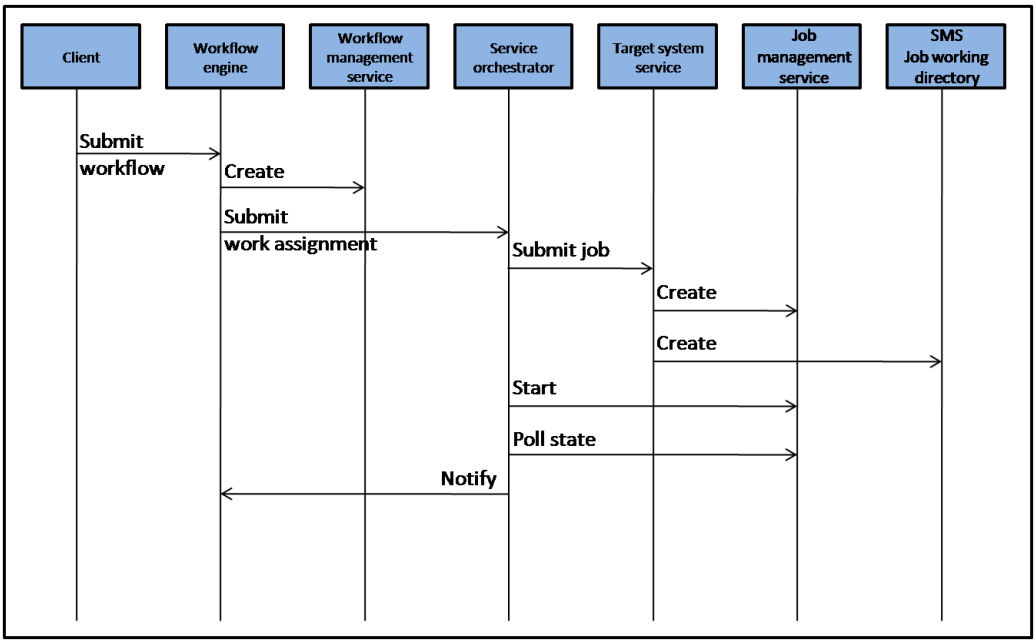


Figure 2.3: Service communication during UNICORE workflow execution

The workflow engine transforms incoming UNICORE workflow descriptions, that are XML documents, into a Java model forming a compound directed graph. It then successively walks along the edges and processes the nodes that represent Grid jobs, workflow variable modifiers and structures like If-Else statements and loops. Nodes that are placed inside loop bodies can be processed multiple times. For Grid jobs this means that multiple output files are created and file name clashes must be avoided. The workflow engine is also capable of keeping track of the nodes' execution states for each of the loop iterations separately.
Grid job nodes that are embedded in the workflow description are transformed into so-called work assignments which are then submitted to the service orchestrator. Each work assignment represents a single job which has not yet been bound to a specific target system.

In addition to the job description formulated in Job Submission Description Language (JSDL) [1], it contains specific meta data, e.g. the address of a storage management service to which all job output files shall be uploaded. The service orchestrator obtains the work assignments and submits the embedded job descriptions to a suitable target system service (TSS) which creates a job management service (JMS) and a storage management service (SMS, representing the job working directory). Next, the service orchestrator starts the job and begins to poll its execution state. Once the job has reached a terminal state (finished or failed), the service orchestrator notifies the workflow engine via a dedicated Web service.
All messages between the workflow engine and the service orchestrator are recorded by a specialised service, the so-called tracer service. Message traces can later be used to reproduce scientific results or observe the state and performance of workflow related services.

2.3 Web application frameworks

A Web application framework is a software framework which allows the development of dynamic Websites and Web applications without handling low level details. Web applications are software modules accessible over a network. Web application frameworks provide tools for code reuse, data persistence, templating and session management. These attributes are useful to divide the Web application frameworks into categories. In the next sections the most important attributes for Web application frameworks are described.

The Model View Controller design pattern
Web application frameworks are often based on variations of the Model View Controller design pattern (MVC pattern [13]). MVC organises an interactive application into three separate units: one for the application model, one for the view and one for the controller. The application model contains the data representation and the business logic. The view is responsible for data visualisation and handling of user input. The view part is often realised by HyperText Markup Language (HTML) descriptions explained later in the section *Web template system*. The controller performs requests on the application model and posts responses to the view layer.
A typical request is to ask for a Web page corresponding to a specific Uniform Resource Locator (URL). A Web page presents a document accessible over the Internet. The connection between URL and Web page is described in the section *URL mapping*.
The response for the request is the content of the Web page. The interconnection described above between the three modules is visualised in Figure 2.4. In most software systems the three modules are not clearly distinguishable from each other. Parts of the software can control business logic and perform requests on it as well. The MVC principle also makes it possible to reuse code, for example view components can be used for different data models.

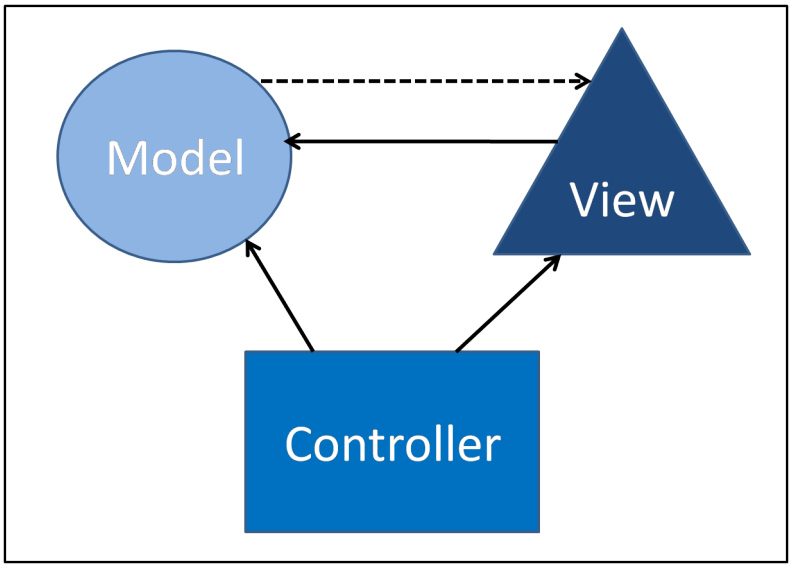


Figure 2.4: Model View Controller design pattern, source [15]

Push and pull based architectures
Framework architectures can be divided into two categories: push and pull. In order to render data after processing, the data can be "pushed" to the view layer. Ruby on Rails[16] is a Web application framework providing a push based architecture. On the other side results can be "pulled" from controllers for rendering. Apache Wicket, described later, realises a pull based architecture.

Database integration
In order to offer high level access to a database, Web application frameworks provide an API for handling changes to the underlying data base schema. Object-relational Web application frameworks permit a mapping between objects and database entity presentation. Additionally some frameworks like Ruby on Rails support database migration tools.

Web template system
A view template is a presentation component which controls the layout. A template is realised by using a markup language like HTML or XML. Different pages can have the same layout using the same view template. In order to connect the view module with the business control module, variables in the template can be used as placeholders for dynamic data provided by the controller.

Security
It is often desirable that users and administrators have different views on pages. In order to restrict access for users, authorisation systems are needed. Web application frameworks like Django[17] or Zope[18] provide security based on access control lists.

[16]Ruby on Rails: http://rubyonrails.org/
[17]Django: http://www.djangoproject.com/
[18]Zope: http://zope2.zope.org/

URL mapping

Each URL request has to be translated for processing and response creation[19]. A URL looks like the following example: http://www.google.com/search?q=url. The URL consists of a protocol name (http), a domain name (www.google.com) or ip address and a port number (which can be omitted and defaults to 80), the path to the resource or the program to be run (search), a query string (q=url), and an optional fragment identifier (specified with #fragment_id). URL mapping can be done in different ways. Web application frameworks like Django use pattern matching by regular expressions. Another technique is URL rewriting, which allows to create simple readable URLs. The example above would then look like this: http://www.google.com/url.
The last technique is to interpret URLs by graph traversal. Zope is a Python based Web application framework where each URL represents an arbitrary deep hierarchy and each URL segment stands for one graph edge.

Ajax

In 2005 the term "Ajax" was coined by James Garret in his famous article "Ajax: A New Approach to Web Applications" [12]. Ajax is a shorthand for "Asynchronous JavaScript and XML". The request/response principle can be handled in different ways. Ajax allows to update page parts and to perform requests asynchronously by the Ajax Engine in the background, which make Web applications more responsive. Unfortunately the replacement of page parts via Ajax can sometimes be a disadvantage for users, who may be wondering why nothing visible happens during the process. For normal requests the page shows a white blank page during the server is still processing the request and updating the whole page. Figure 2.5 shows the comparison between handling of normal (left hand side) and Ajax requests (right hand side).

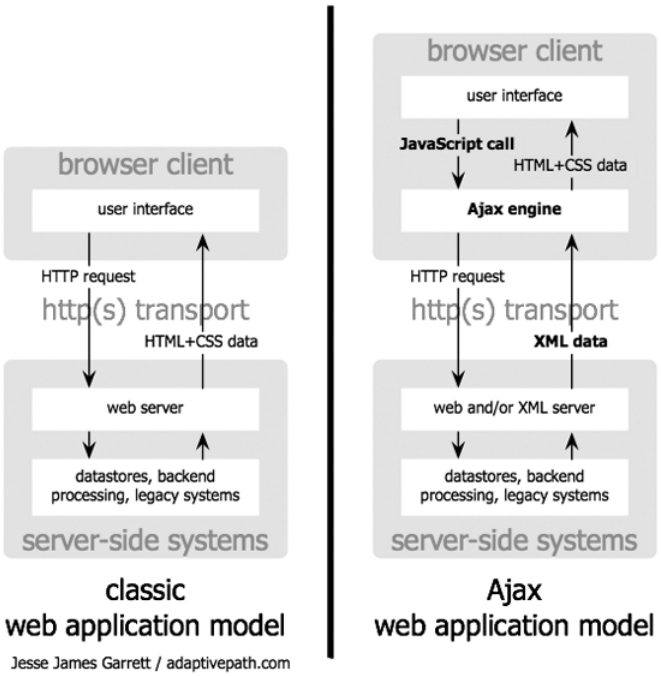


Figure 2.5: Comparison of normal and Ajax requests

[19]URL mapping: http://docs.repoze.org/bfg/narr/urlmapping.html

REST/SOAP
Simple Object Access Protocol (SOAP), based on XML, is a protocol for the exchange of structured information. In contrast to SOAP, Representational State Transfer (REST) is an architecture style. The decision for using SOAP or REST based approaches depends on system requirements. SOAP has the big advantage of coupling components where REST benefits lie in the potential scalability [20]. Ajax is using SOAP or REST for the exchange of request and response documents. The response document can be described by HTML, XML or JavaScript Object Notation (JSON). JSON[20] is a text format and a data-interchange language, which uses conventions that are familiar to Java, JavaScript and many others.

2.4 Apache Wicket

This master thesis involves the development of a Web application by using the open source project Apache Wicket. Apache Wicket has been selected for the reasons explained below. The first version of Wicket was released in 2004 by Jonathan Locke. Wicket is known as a Web application framework which allows an efficient and easy development of Web applications in the object-oriented programming language Java. It merges stateless HTTP with stateful Java programming. The book "Wicket in Action [6]" gives a clear description of how to use Wicket and what its benefits are. The following list shows the main concepts in Wicket, which are important for the scope of this thesis.

- Wicket allows the development of dynamic Web applications, which are represented by application objects containing all application properties.
- Wicket uses sessions which last from user login to user logout in order to handle user interaction.
- As explained in the previous chapter, the interaction with the user is realised by requests and responses. Every request contains parameters like the requested URL. Responses are needed to generate the corresponding answer. In Wicket the request/response handling is done by a request cycle. Every request is processed by a unique request cycle.

Like most Web application frameworks, Wicket is based on the MVC principle. The following Figure 2.6 shows the MVC interconnection in Wicket based on a simple *echo application* example. Parts of the example code for the *echo application* example are obtained from the "Wicket in Action" book. The *echo application* example takes a user input like "Say Hello" and returns it on a label component after clicking the button "Set Text".
A model represents the data, in this case the string "Say Hello". Wicket provides different types of models such as detachable, resource and property models. Detachable models release most of their object graph retaining just enough detail (typically a unique identifier) that can be used to reconstruct the objects. Thus the size of the serialised objects will be reduced. Resource and property models are dynamic allowing the re-evaluation of the model object each call.

[20] JSON: http://www.json.org/

Resource models integrate expressions stored in a property file application or component specific whereas property models allow to access a particular property of its associated model object. Besides specific model types it is also possible to use static models whose values never change. Such static models are often used for labels.
The view module renders the data in a label component using HTML descriptions. The controller handles the processing between model and view. Wicket organises the view and controller parts in so-called components which are responsible for visualisation, but also for receiving user input and updating the view part. Wicket components differ in their behaviour. Each piece of behaviour is encapsulated by a behaviour class. A piece of behaviour can be a self-updating process every two seconds or the view updating process when a user clicks on a button component, as in the *echo application* example.

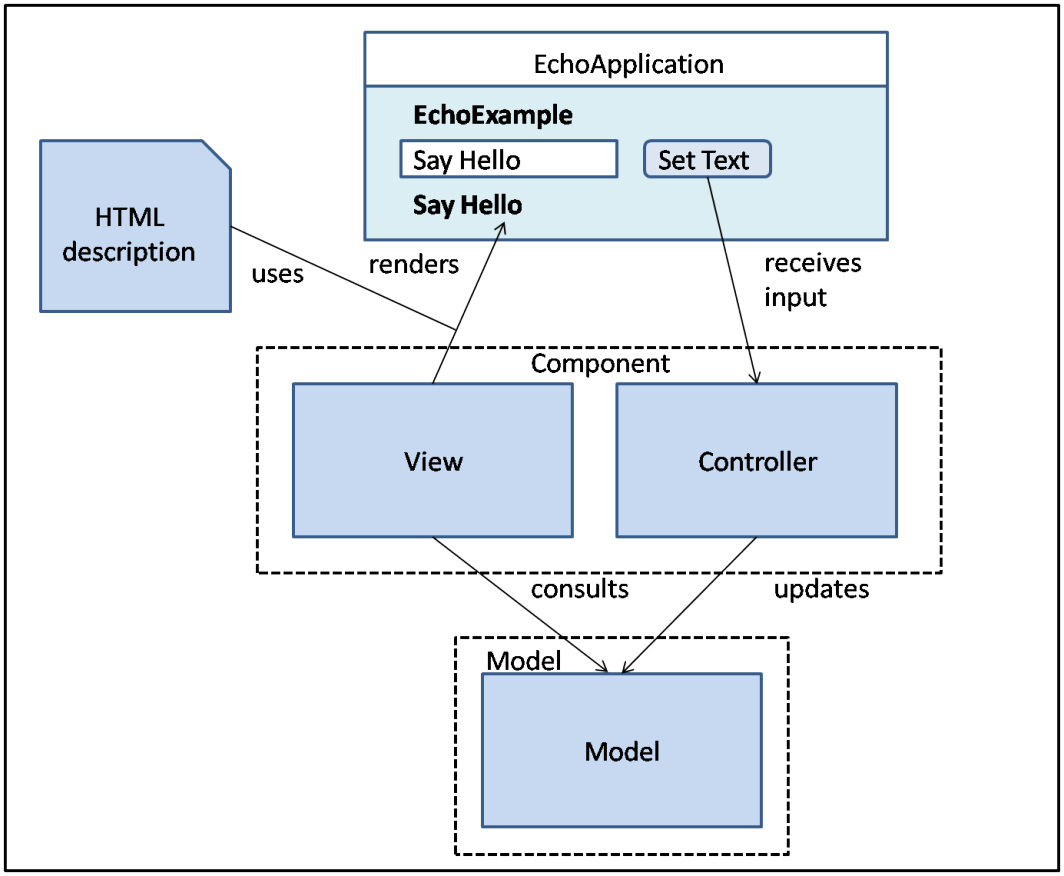


Figure 2.6: MVC principle in Apache Wicket

One of the major benefits of Wicket is its state management, which is done transparently behind the scenes. For a minimal Wicket Web application, at least four parts are needed: an application class, a Web page class, a corresponding HTML description and a configuration file. These parts are explained for the *echo application* example in the next section, starting with the specification of the application class.

Application class

Every Web application implementation consists of an application Java class which presents the entry point for the application. For the *echo application* example Listing 2.1 shows the application class, which defines the main Web page, the so-called echo page, described in the Web page part.

Listing 2.1: Implementation of the application Java class

```
public class EchoApplication extends WebApplication {
    public EchoApplication() {
  }

  @Override
  public Class getHomePage() {
    return EchoPage.class;
  }
}
```

Web page

Web pages consist of Wicket components. A component is a Java class that is supplemented by behaviour classes. Examples of components are labels or text fields as shown in the *echo application* example. A label is used to show non editable text e.g. by using a static model whereas a text field component can receive user input. In order to process the input, a key pressed event or a button component is needed. Listing 2.2 shows the echo page implementation. Both a text field and button are added to a form component whereas a label is added to the page. Forms allow users to enter data which are sent to a server with either POST or GET method calls. The Wicket default POST method is often used for long URLs during file uploads. The GET method is restricted to a URL length but useful for readable and reusable URLs. In the *echo application* example the processing is done in the following way: If the button was pressed by the user, the behaviour of the button and the surrounding form will be invoked. The submission process comprises three parts:

- Wicket first checks whether the input value for a component is required or not. If it is empty but mandatory, no process will be started and the user gets a feedback message.
- Validators are used to validate the input value and are explained later in this chapter.
- The components' submission methods are called. In this example the user input string is used as new model object for the label and the whole page will be rendered again.

If one of these steps fails, Wicket returns feedback messages to the user. These messages can be placed on a panel component which acts as a container element.

Listing 2.2: Implementation of the echo page Java class

```
public class EchoPage extends Webpage{
  private Label label;
  private TextField field;

  public EchoPage(){
    Form form=new Form("form");
    field=new TextField("field", new Model(""));
    form.add(field);
    form.add(new Button("button"){
      @Override
      public void onSubmit(){
        String value=(String)field.getModelObject();
        label.setModelObject(value);
        field.setModelObject("");
      }
    };
    add(form);
    add(label=new Label("message" new Model("")));
  }
}
```

HTML

The view part is realised using HTML descriptions. In contrast to other presentation templates the page hierarchy is important. Listing 2.3 shows the HTML description for the example Java Web page above. The page contains a form which contains a text field and a button. The HTML structure must reflect the hierarchy of the according page class.

Listing 2.3: Implementation of the echo page HTML description

```
<html>
<head><title>Echo Application</title></head>
<body>
  <h1>Echo Example</h1>
  <form wicket:id="form">
    <input wicket:id="field" type="text"/>
    <input wicket:id="button" type="submit" value="Set␣text"/>
  </form>
  <p wicket:id="message">[message]</p>
</body>
</html>
```

Configuration file

In order to configure the Web application, it is required to create a web.xml file and define specific parameters like the Web application class. Listing 2.4 shows the configuration for the *echo application* example. The filter specification defines how requests should be processed via a servlet filter by looking at the filter's mapping parameters.

Listing 2.4: Configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun␣Microsystems,␣Inc.//DTD␣Web␣Application␣2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Wicket Example</display-name>
  <filter>
    <filter-name>EchoApplication</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter
    </filter-class>
    <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>org.apache.wicket.examples.echo.EchoApplication
    </param-value>
    </init-param>
  </filter>
  <filter-mapping>
      <filter-name>EchoApplication</filter-name>
      <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

Validators
Validators are used to check the user input, ensuring that only correct data will be processed. Wicket provides many validators, e.g. a number validator or a string validator, which must be registered for a component and have attributes which declare the allowed values. A number validator can be specified by setting a minimum and maximum number whereas string validators have attributes like the maximum or minimum string length. After receiving the user input, all validators are consulted. If the input is valid, it is pushed to the models and finally the method for submission will be invoked. For example in the *echo application* a string validator could be specified with a minimum length of four characters. If the user then enters a word with three characters, he or she will get an error message saying that his input was invalid.

Wicket and Ajax
Ajax allows the reload of page parts instead of the whole page. Wicket comprises Ajax specific components like an Ajax button which reacts to Ajax requests. In the *echo application* the Ajax button could react on an On-Click event by updating the label component.

Including resources
In addition to components, Wicket provides resource objects which are responsible for CSS style sheets [9], images and Java script files. It is also possible to integrate third party libraries or to use dynamic resources. CSS is a style-sheet format which influences the look of components and can be integrated in the HTML description or be injected via resource reference. Images and files can also be integrated as resource objects, such as link icons. The following piece of code shows a link component which adds an image including a resource object with a specific class and name.

Listing 2.5: Example for including a resource object

```
link.add(new Image("icon", new ResourceReference(List.class, "icon.gif")));
```

2.5 Groovy

As explained in the previous section every Wicket container component is realised by a Java class. For container components like panel or page a corresponding HTML description is required. During the export function panel components for the visualisation of exported workflow structures in the Web application must be generated. In order to avoid Java compiling with complex requirements like JDK (Java Development Kit) the Java byte code will be generated by using Groovy files. Groovy was born in 2004. It is a dynamic language for Java influenced by programming languages like Python, Ruby, and Smalltalk [7]. Groovy can be seen as a rich feature for Java. The syntax is similar to Java but has been simplified to fit the needs of a scripting language. The most important differences between Java and Groovy are shown in the following list[21]:

- Every Java line of code ends with a semicolon whereas in Groovy the use of semicolons is optional.
- In Java it is possible to restrict the visibility of a class. The default declaration for a class is *protected* meaning that not all classes are able to see it. In Groovy the default is *public*, so the class is visible to all other classes.
- In Groovy, return types don't need to be explicitly declared, Groovy also allows dynamic typing.
- Compiling errors are not visible in Groovy.
- Instead of anonymous inner classes in Java, Groovy uses closures. A closure is one or more program statements enclosed in curly bracket.

These are the most important differences. Listing 2.6 illustrates some of these differences by comparing Java code to Groovy code for the well-known "Hello World" example [7].

Listing 2.6: Groovy example

```
//Hello World example in Java
class HelloWorld {
   public static void main( String[] args ){
      System.out.println("Hello␣World!");
   }
}

//Hello World example in Groovy
class HelloWorld {
   def greet( name ){
      "Hello␣${name}!"
   }
}

def hw = new HelloWorld()
println hw.greet("Groovy")
```

[21] Groovy and Java comparison: http://Groovy.codehaus.org/Differences+from+Java

Groovy is an object-oriented language such as Java. Thus both programming languages contain a class definition. The *main* method is required in Java to define the entry point for processing. It is declared with semicolons, the type string for input parameters and the return type *void*. Because of the script syntax Groovy needs no *main* method and the statements are processed top-down. The declaration of variables and methods are labeled with the key word *def*. In order to keep the script syntax more readable, semicolons and type declarations are omitted as well as return types.

Chapter 3

Requirement analysis

Users working with workflow templates can be classified into two categories: Grid workflow experts can provide workflow templates based on pre-defined workflow structures and end-users want to change template parameter values, submit and monitor workflows in a simple user interface. Both scenarios have functional requirements which are explained in the next two sections beginning with workflow template creation.

3.1 Workflow template creation

The first task is the creation of a workflow with the URC workflow editor. For the generation of workflow templates, an export function is required which provides a choice of possible template parameters as well as the selection of meta information (Figure 3.1). There are currently six different parameter types:

- Primitive job parameters: Grid jobs lead to the execution of applications, which require command line arguments and environment variables to be set. These parameters are called primitive job parameters as they are simple textual parameters, most often representing key words or numbers.
- Files and file sets: Grid jobs often require input files or file sets containing the actual scientific data. File sets can also be iterated in UNICORE workflows using the For-Each construct.
- Workflow variables: UNICORE workflow activities (the nodes in the workflow graph) can declare typed variables that can be modified and evaluated during workflow execution. These variables can serve as input parameters for jobs or steer the flow of control.
- Variable modifiers: UNICORE workflow activities can define variable modifiers that are used to change the values of workflow variables. To this end a workflow variable name and a modifier expression are specified. While it usually does not make sense to interpret the variable name as a template parameter, the end-user should be allowed to set the modifier expression in the workflow template. This can be useful e.g. for influencing the number of iterations in loops.

- Condition: There are four workflow structures may apply conditions: If-Else structures decide whether to process the if or else branch based on a condition, Repeat, While and For-Each loops have terminal conditions determining when the loop is finished. There are two types of conditions in the URC workflow editor: A comparison involving the exit status of a pre-defined job and a comparison involving a workflow variable value.

Workflow engine services process workflows autonomously. This allows users to go online and offline arbitrarily during workflow execution but it implies that local input files must be available online. Therefore meta information comprises a storage reference for uploading local input files.
The result of the export function is a deployable workflow template archive which comprises among others a workflow template and a pluggable template panel which allows the substitution of template parameters.

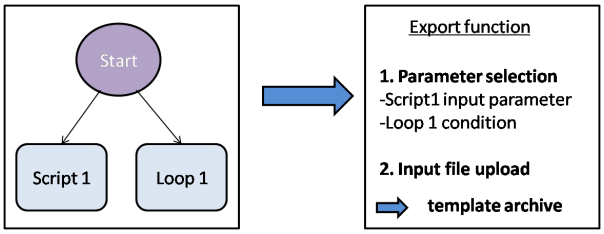


Figure 3.1: Export function

3.2 Web application functions

The exported workflow template archives can be uploaded to the Web application which should provide functions for workflow submission and monitoring. Furthermore a worklow template repository should store workflow templates and should allow access to these templates via Java interfaces. By persistently storing all of the end-user's workflow templates as well as the derived workflow executions previous scientific experiments and results can be reproduced easily.

3.2.1 Workflow template repository

The Web application must be able to manage uploaded workflow templates as well as their manipulation by end-users. In order to manage a workflow template repository components for uploading archives, manipulating template parameters and deleting templates must be provided.

Uploading workflow template archives
The end-user should be able to upload the exported workflow template archives to the Web application. Therefore the Web application needs to offer a function for uploading archives.

Uploaded archives will be stored in a directory at the Web application server and will be accessible through the workflow template repository.
Figure 3.2 shows on the left hand side three uploaded workflow templates displayed in a list structure. The size of this list should be limited in order to respect performance reasons. The displayed names are the names of the exported workflow template archives.

Manipulation of template parameters
Once uploaded, the workflow template archive is analysed in order to identify the required files. A pluggable template panel is required to produce a view of the workflow structure and parameters. Figure 3.2 shows on the right hand side an example template panel of the chosen workflow template (called *Template 3*). The template panel should allow users to change template parameter values, to reset them to the default value or to submit the finalised workflow templates. The example panel shows the selection of a loop structure (called *Loop 1*) and corresponding parameters (Counter, Modifier, Condition) in order to change their value. Similarly to the export function, local input files must be available online if they are to be used for job stage-ins. Therefore new files are uploaded to a directory on the Web application server, the so-called workflow working directory. The workflow working directory contains besides input files for the workflow template all produced output files of executed workflows. This way, when fetching workflow outcomes via the Web application, files can directly be served from the workflow working directory which is a local directory. This is faster than fetching files from the remote job working directory.

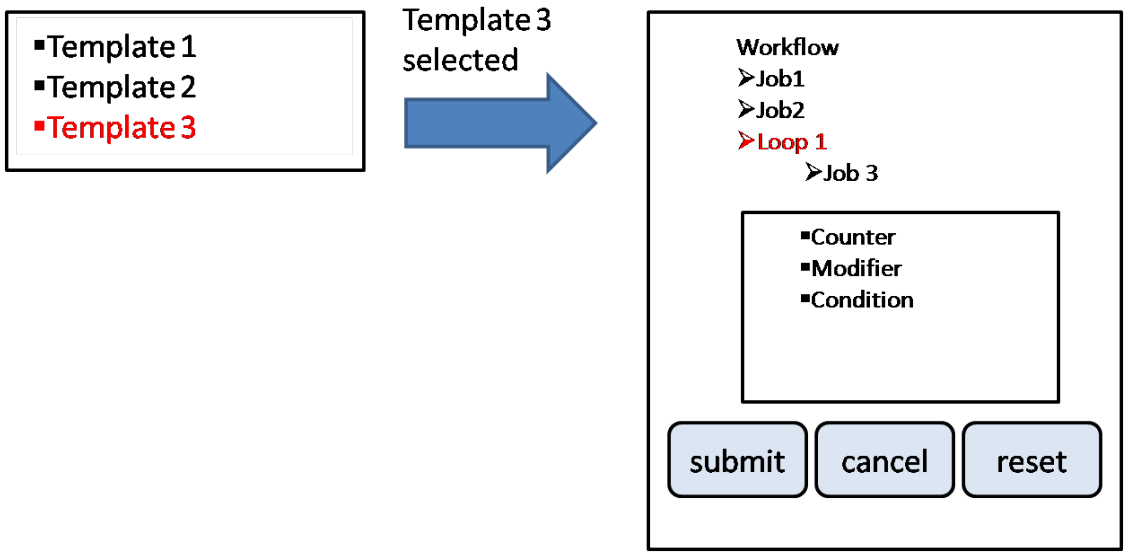


Figure 3.2: Workflow parametrisation function

Deleting workflow templates
It is important to note that submitted workflows still depend on the workflow template they are derived from. Hence, if a template is deleted, the corresponding submitted workflows will be erased as well.

3.2.2 Execution and monitoring of workflows

The Web application should provide functions similar to the basic workflow functions in the URC which involve submission, cancelling, monitoring and fetching output of workflows.

Submitting a finalised workflow template
When the workflow template has been completed all along, the user should be able to submit the finalised workflow template to the UNICORE workflow system. Each submitted workflow is displayed beneath the corresponding workflow template.

Deleting submitted workflows
Each submitted workflow can be deleted by user interaction. If a submitted workflow is deleted, the corresponding template archive will not.

Cancelling submitted workflows
The end-user should be able to cancel the execution of a submitted workflow at any time. Aborted workflows are marked as such in the user interface.

Monitoring submitted workflows
In order to monitor a submitted workflow, the current execution state is fetched from the UNICORE workflow system. The status of submitted workflows should be visualised. One way of visualising a submitted workflow is presented in Figure 3.3 on the right hand side. As soon as the workflow fails or finishes successfully, the user should get feedback. The figure shows the monitoring view of the submitted workflow *Template 3-submitted1*. The submitted workflow is presented beneath the derived workflow template *Template 3* in the list on the left hand side.

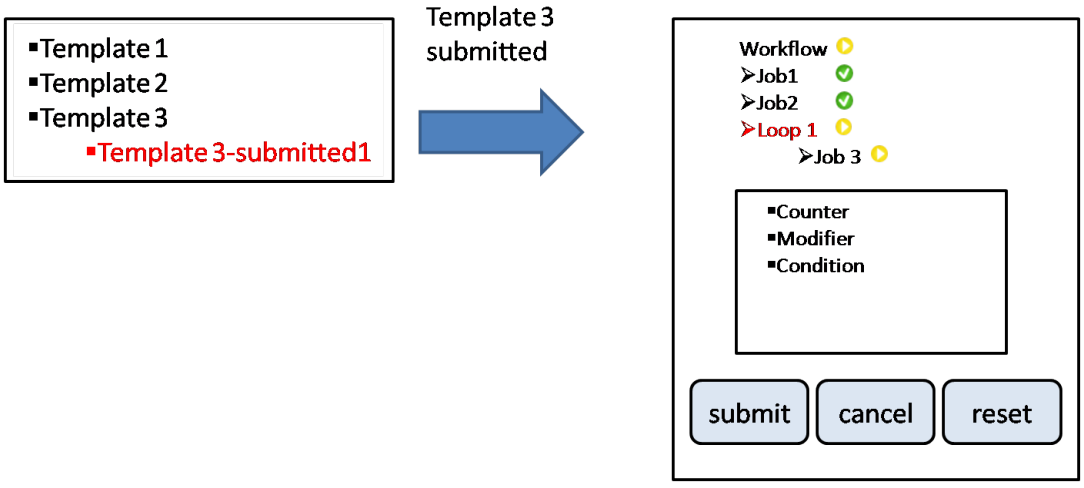


Figure 3.3: Execution and monitoring function

Downloading output files

After successful or failed workflow execution, information about any produced output files should be presented and the user should be able to download output files via the Web application. Figure 3.4 visualises a possible download interface for the selected output file *stdout* of *Job 3*.

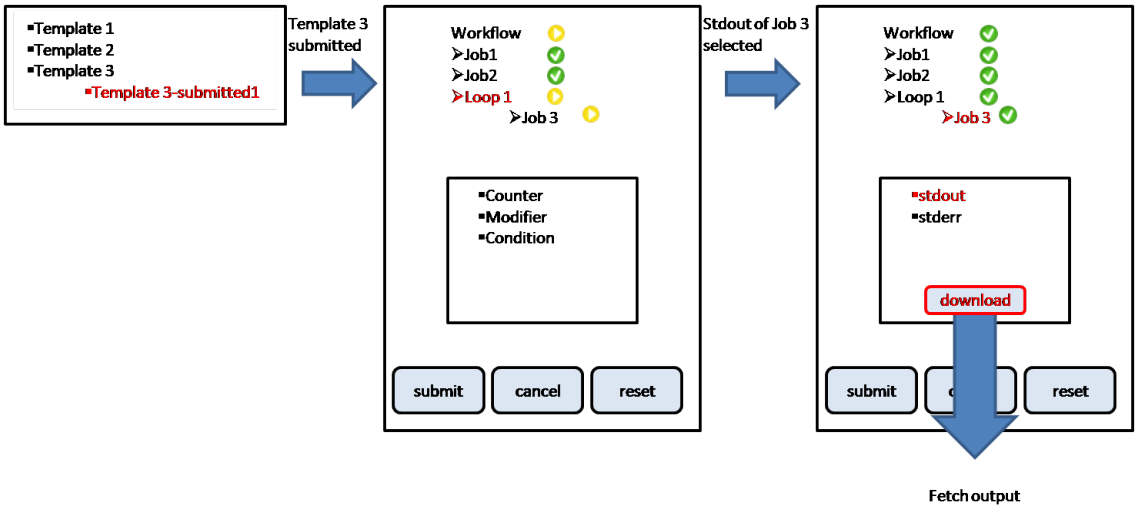


Figure 3.4: Fetch output function

Chapter 4

System architecture

Both scenarios, workflow template creation and management via a Web application, are integrated in one concept. Figure 4.1 presents an overview of the interplay between the two parts. A Grid workflow expert provides workflow templates, based on workflow structures, by using the export function in the URC (top left in Figure 4.1). The produced workflow template archive (called *Blender.jar* in the example) will be uploaded to the Web application (bottom right in Figure 4.1).

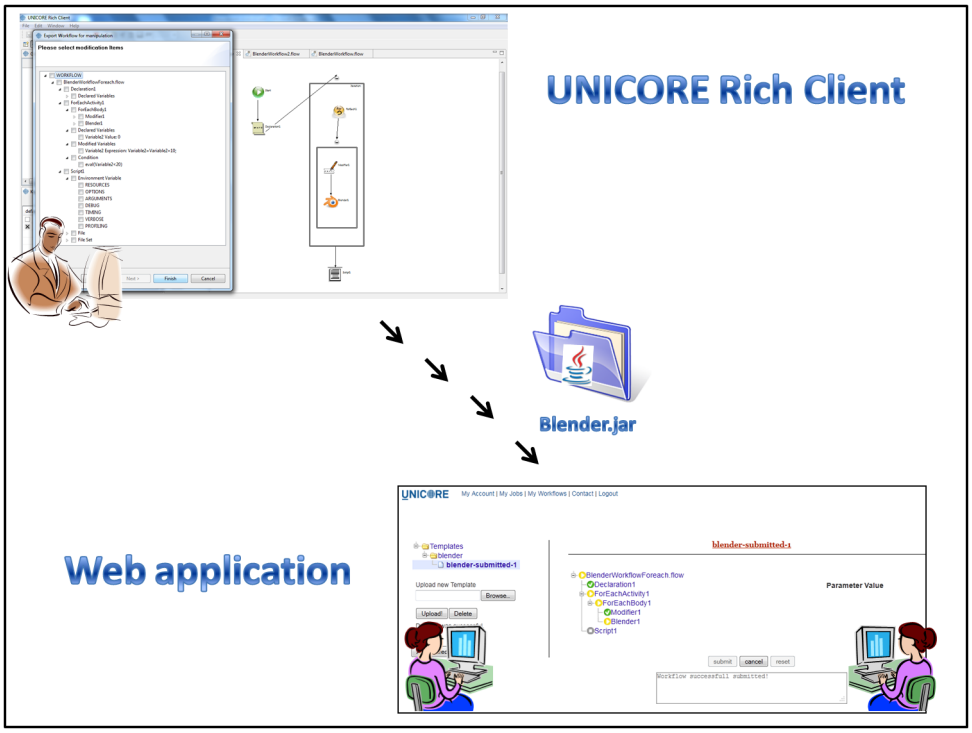


Figure 4.1: Overview: Export function and Web application

All export steps, including selection of template parameters and specification of meta information, are combined in one function, which is implemented as an Eclipse plugin and thereby extends the URC workflow editor. Figure 4.2 shows the component diagram for the new export function. A component diagram visualises structural relationships between components in a system. Components highlighted in red are part of the development in the scope of this work and present modular parts which encapsulate their contents.

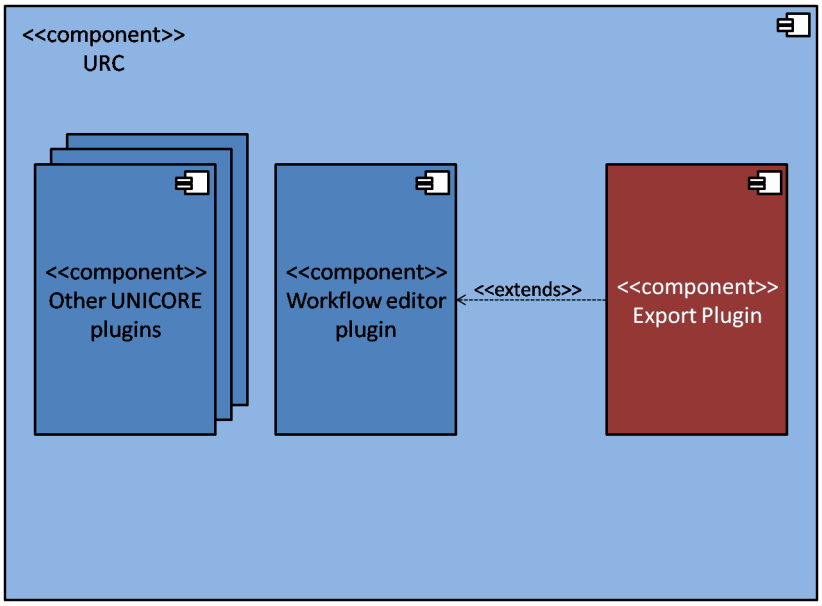


Figure 4.2: Component diagram for the export function

The component diagram in Figure 4.3 presents the relationships between the components in the Web application, which is based on the current UNICORE Web client design, and their connection to components outside. The Web application consists of a set of Web pages; besides the workflow page it includes the login page and the job page. All pages need access to the user data base to get information about the acting user. The workflow page extends the current UNICORE Web client and provides a workflow template repository as well as operations for workflow processing. The workflow page employs pluggable template panels which provide a workflow specific user interface for substituting template parameters. After finalising workflow templates, they can be submitted to the UNICORE workflow system. Subsequently, the workflow page contacts the workflow system in order to monitor the workflow execution. A dedicated workflow working directory is required to make the end-user's input files available to the UNICORE workflow system. This directory must be accessible remotely via a UNICORE storage management service. Figure 4.4 shows a deployment diagram for the client components (URC, browser) and the server components (Web application and workflow working directory). For performance reasons, the workflow working directory should be deployed on the same system as the Web application. The URC and the browser can run on any client machine.
The design and implementation of the export function and the Web application are explained in detail in the next chapters.

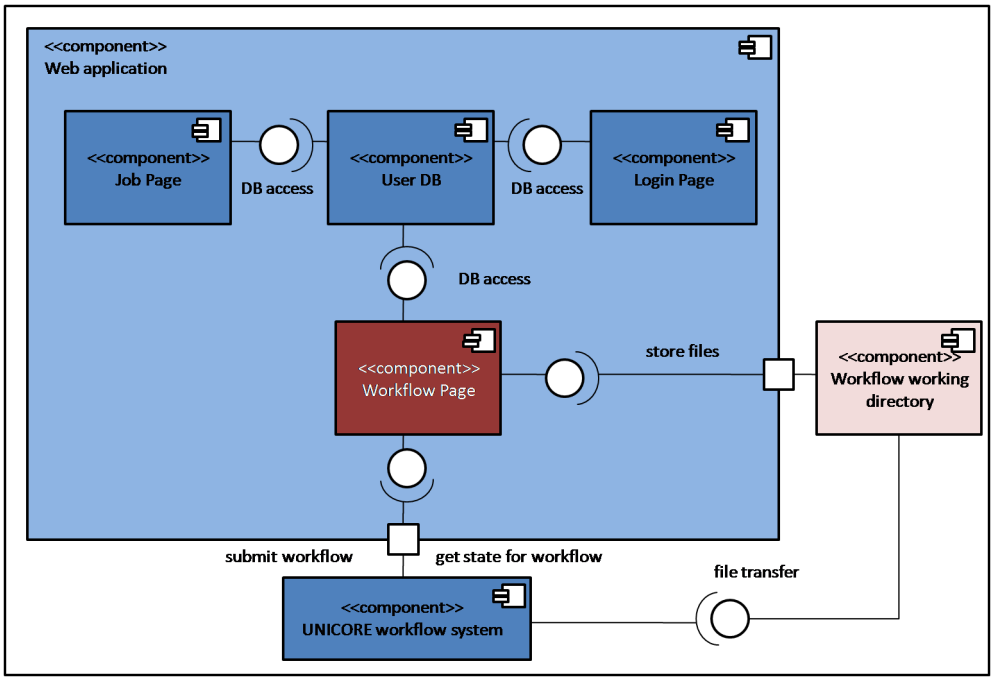


Figure 4.3: Component diagram for the Web application

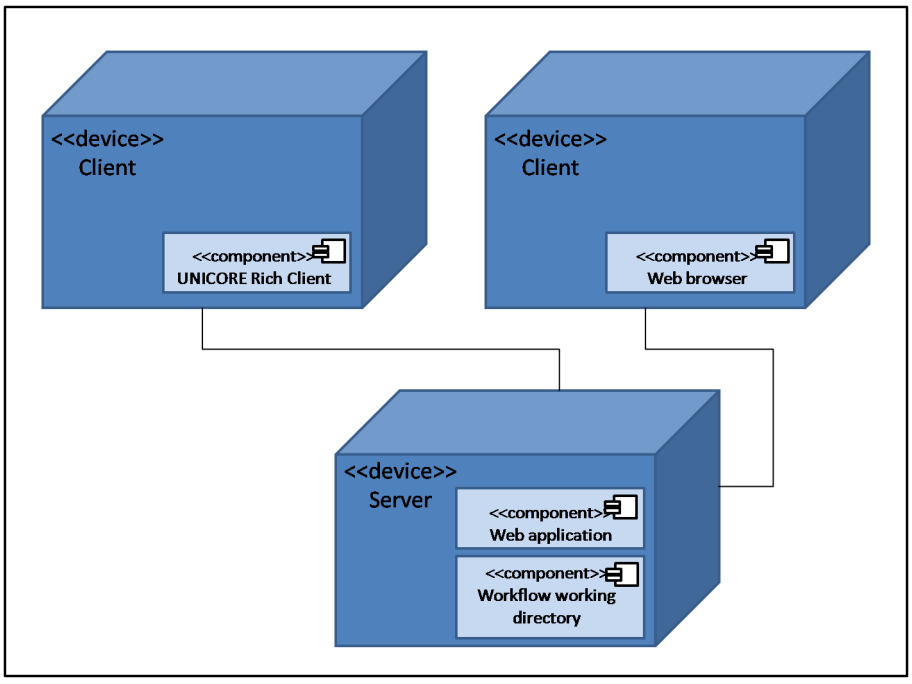


Figure 4.4: Deployment of involved components

Chapter 5

System design

5.1 Export function

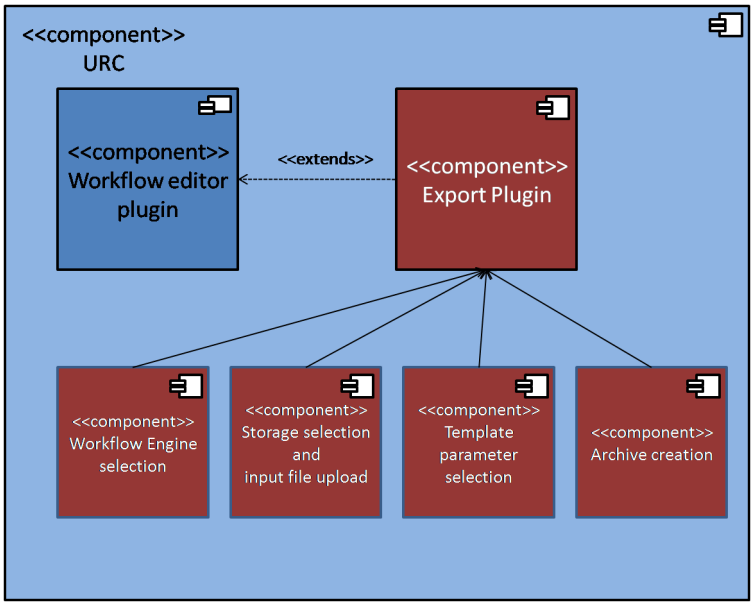


Figure 5.1: Component diagram for the export steps

This section starts with the design description of the export function as visualised in the component diagram in Figure 5.1. The export function comprises four steps: workflow engine selection, storage selection and input file upload, template parameter selection, and archive creation. For template parameter selection, an analysis of all existing parameters embedded in workflow activities and structures is required. In order to generate a workflow template, a template object model has to be created which is explained at the end of this chapter. The archive comprises the template object model, the XML workflow template, and files for the corresponding pluggable template panels.

5.1.1 Workflow engine and storage selection

The first two steps of the export function are rather simple and thus only described in detail in the implementation chapter.

5.1.2 Template parameter selection

Template parameters will be substituted with placeholders in the XML workflow template. Currently there are six different parameter types: primitive job parameters, files, file sets, workflow variables, variable modifiers and terminal conditions, which are used by different workflow structures:

- GridBean activities are application plugins in the URC and offer a graphical user interface for the definition of job descriptions. The job description is formulated in JSDL. GridBean activities contain different job-specific primitive parameters, input and output files, and input and output file sets.
- A Variable Declaration activity declares a workflow variable with a specified initial value, whose type can be integer, float or string.
- The Variable Modifier activity can change any defined variable value during workflow execution by evaluating an expression.
- The If-Else construct has two subgraphs, the if body and the else body. Which of the subgraphs is executed depends on the evaluation of a condition.
- A While loop processes its containing tasks as long as a specified condition is met. Loops are described by an iteration variable, a variable modifier that is executed each iteration and a condition (that is often based on the iteration variable).
- The Repeat loop is iterated until the specified condition is met.
- In contrast to other loops, the iterations of the For-Each loop can be executed in parallel. Parallel execution is only possible, if the number of iterations is known beforehand, which does not have to be true for While or Repeat loops. For-Each loops can iterate over variable values or files in a file set.
- The Group structure merely acts as a container and therefore has no parameters.

Figure 5.2 shows a workflow created with the URC workflow editor; it contains a While loop with an embedded script job. The job is executed five times. Since the counter (*WhileActivity1_Iteration_Counter1*) is initialised with one and increased by one in each iteration through the modification expression *WhileActivity1_Iteration_Counter1++* as long as the counter value is less or equal than five.

The Script GridBean contains eight parameters: *OPTIONS*, *ARGUMENTS*, *DEBUG*, *TIMING*, *VERBOSE*, *PROFILING*, a *SOURCE* file and an *INPUT* file set.
The result of the parameter analysis is a set of primitive job parameters, input files, input file sets, workflow variables, variable modifiers and conditions that can be selected as possible template parameters.

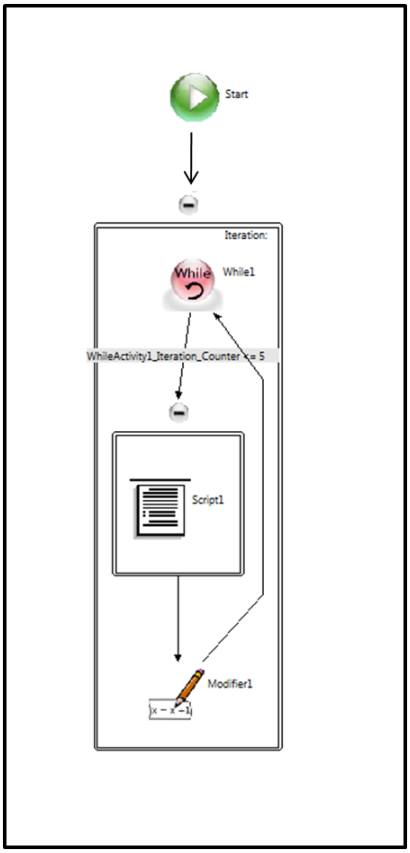


Figure 5.2: Example workflow

5.1.3 Archive creation and the template object model

In order to get quick access to template parameters and to visualise the workflow structure, the workflow model will be converted into a template object model, which is described in the following by using a simple example.
Figure 5.3 shows the relation between the workflow structure, created in the URC workflow editor and the template object model. As shown on the left hand side of the figure, the workflow consists of a For-Each loop and a Group construct. Both tasks will be processed in parallel. This is illustrated by two transitions, one from the Start activity to the For-Each loop and one from the Start activity to the Group construct. The For-Each loop contains a Script GridBean activity whereas the Group construct comprises a Script GridBean and a Blender GridBean activity. The template object model is structured similarly, but compared to the object model of the URC workflow editor it is simplified slightly.

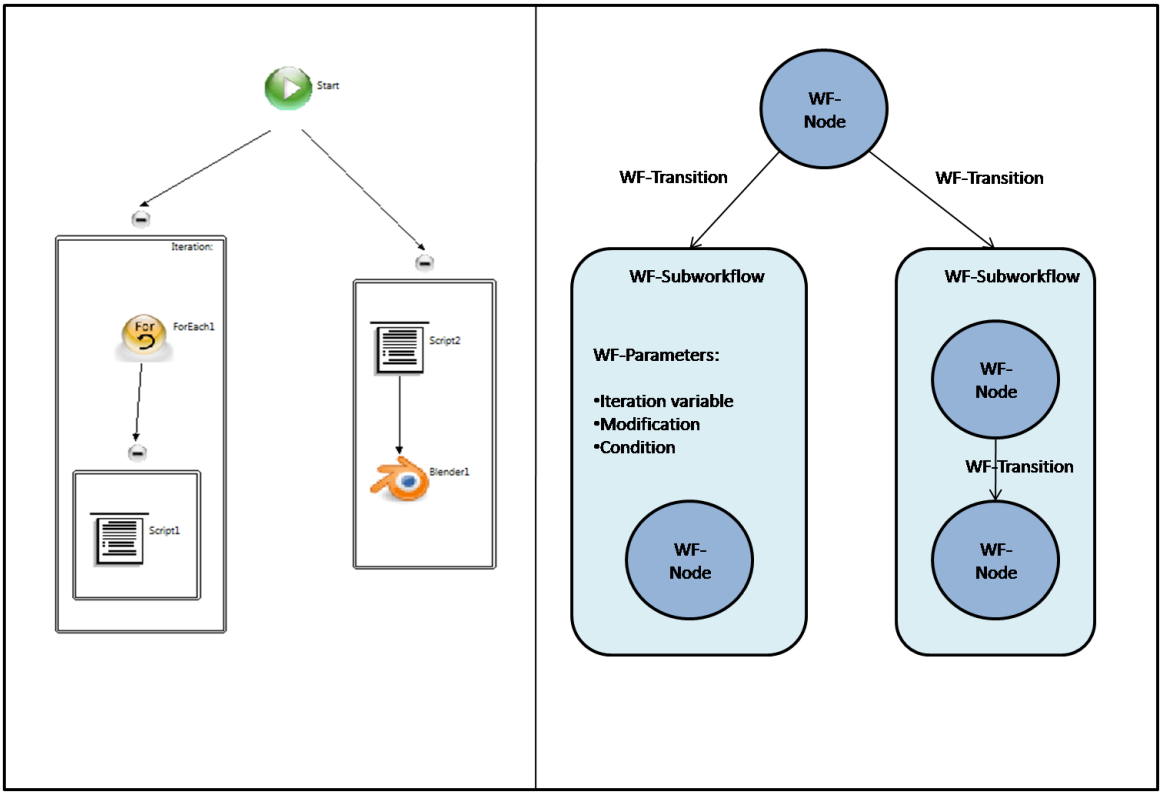


Figure 5.3: Converting the workflow model into a template object model

WF-Parameter
All parameter types are represented by `WF-Parameter` elements. For example, the For-Each loop has three parameters, an iteration variable, a modifier and a terminal condition. In contrast, the Script GridBean activity has primitive job parameters (such as command line arguments) and input files.

WF-Node
All atomic activities like GridBean, Start, Variable Declaration and Variable Modifier activities are modelled as `WF-Node` parts, which contain `WF-Parameter` entries. The Start activity has no parameters, because it only symbolises the beginning of the workflow. Variable Declaration and Variable Modifier activities contain one parameter each, the variable and the modifying expression, respectively. GridBean activities can have many parameters like primitive job parameters, files or file sets.

WF-Transition
`WF-Transition` elements define the flow of control between activities and have no `WF-Parameter` elements. The source of a `WF-Transition` must be processed before its target during the actual workflow execution.

WF-Subworkflow
`WF-Subworkflow` elements represent complex structures like the For-Each, Repeat and While loop constructs. Groups and the If-Else constructs are represented by `WF-Subworkflow` elements, too. `WF-Subworkflow` may comprise parameters which are converted into `WF-Parameter` elements.

5.2 Web application

This chapter describes the software design of the Web application which is based on the UNICORE Web client. The Web application is developed on top of the Apache Wicket Web application framework. The newly developed workflow page extends the current UNICORE Web client design with a new “My Workflows” link.
The diagram in Figure 5.4 presents the connection between components of the workflow page and components outside, e.g. the communication between the submission and monitoring components and the UNICORE workflow system. The workflow page contains four components: the workflow template repository, the workflow parametrisation, the submission component and the monitoring component.

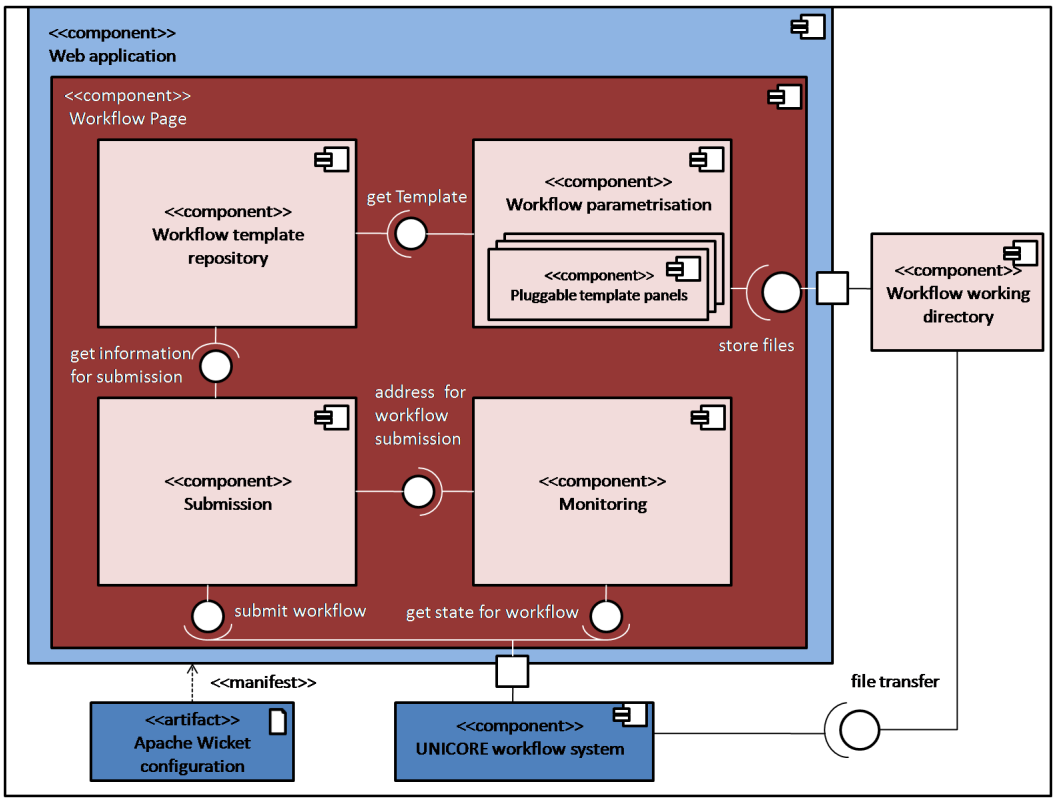


Figure 5.4: Component diagram for the Web application

All workflow template archives are uploaded to the Web application and stored in the workflow template repository which provides interfaces and access to workflow templates for the workflow parametrisation component. The workflow parametrisation component uses pluggable template panels including parameter forms for substituting template parameter values. Pluggable workflow templates can also be added and deleted during the runtime of the Web application. The included parameter forms present all workflow parameters in a user interface. The presentation of workflow parameters depends on the parameter type: primitive parameters, files and file sets. New files are stored inside the workflow working directory.

The pluggable template panels are deployed by uploading workflow template archives generated as a result of the export function. They are modelled with Wicket container components. For each container component, Wicket requires a Java class and an HTML snippet defining the containment hierarchy. These two files must be present in the workflow template archive, which means they must be created by the export function. While creating the HTML snippet is unproblematic, compiling a Java class requires the presence of a Java Development Kit (JDK). In order to avoid this dependency, the Groovy scripting language was chosen for dynamically creating Java classes without compilation.
The workflow template repository provides interfaces for the submission component such as access to submission information like the selected workflow engine. In order to submit a workflow, the workflow template needs to be filled with values from the parameter form. After submission, the submission component provides the address of the newly created workflow management service to the monitoring component which wants to get information about the workflow state from the UNICORE workflow system.
Figure 5.5 shows the interplay between the UNICORE workflow system (see Chapter 2.2.2), the Web application and the workflow working directory (which is a normal UNICORE storage management service, SMS).

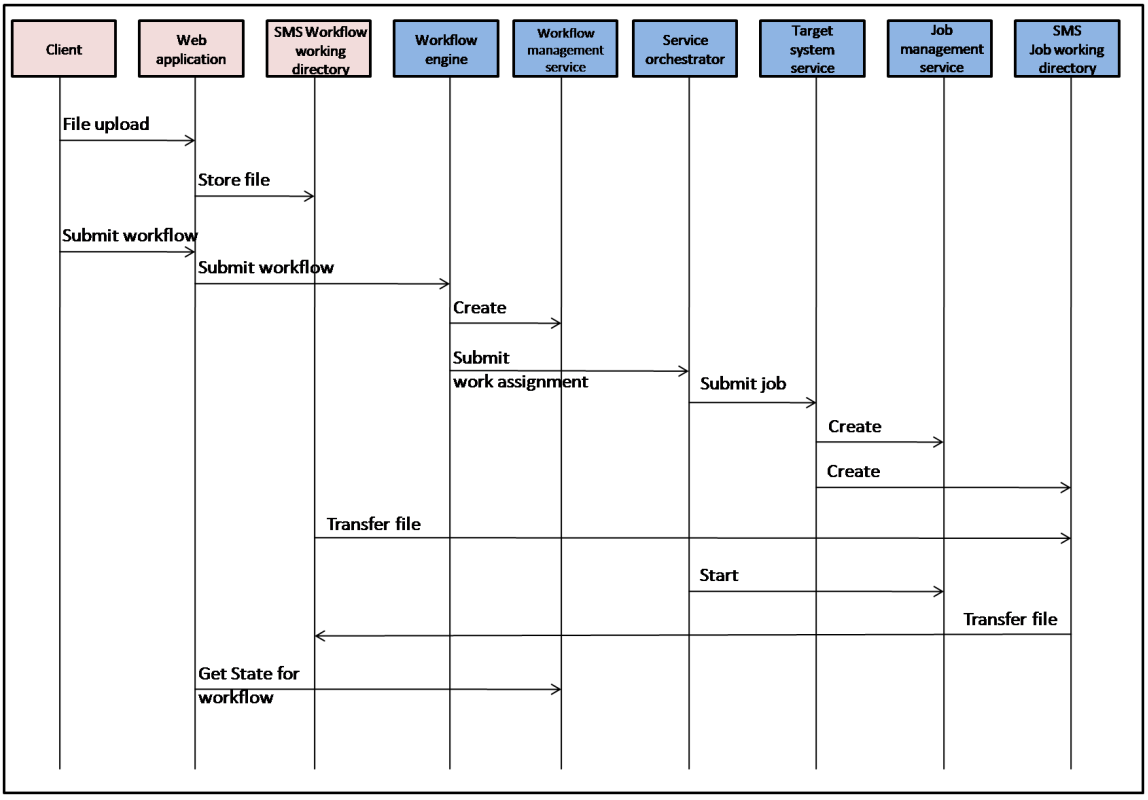


Figure 5.5: Interplay of components during the execution of a simple workflow containing a single Grid job

The simplified sequence diagram visualises the process of a simple workflow consisting of one job which has one input and one output file. The input file is marked as template parameter and thus the client allows to exchange this file value. The user uploads a new file via the Web application which is stored on the workflow working directory.

Next the finalised workflow template is submitted via the Web application. The workflow engine receives the workflow and submits a work assignment (corresponding to the single job) to the service orchestrator. The service orchestrator forwards the embedded job description to a target system service which creates a new job management service and a new storage management service representing the job working directory. Before the service orchestrator starts the job, the input file is transferred from the workflow working directory to the job working directory. All job output files are transferred from the job working directory back to the workflow working directory for persistent warehousing of all workflow outcomes. The Web application interacts with the workflow management system in order to retrieve the workflow state and visualise it for the user.

Chapter 6

Implementation

6.1 Export function

The export function is accessible through a new menu item in the URC workflow editor. The export steps are described in detail in the next subsections. The steps, illustrated in Figure 6.1, comprise the workflow engine selection, storage selection and input file upload, template parameter selection and archive creation.

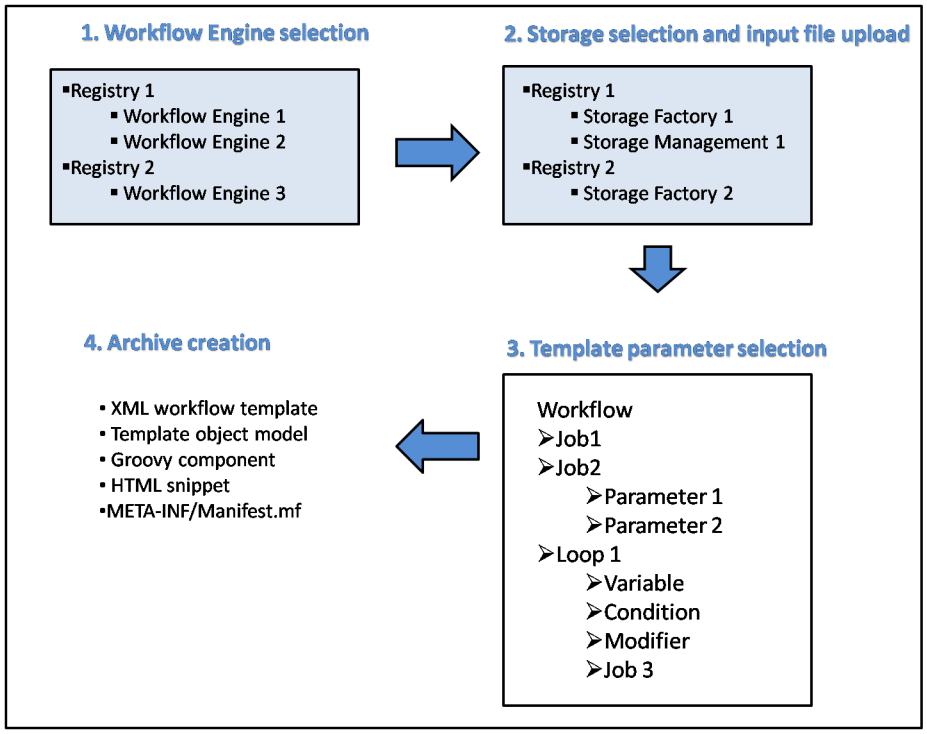


Figure 6.1: Export process

6.1.1 Workflow engine selection

The Grid workflow expert has the experience to decide which workflow engine is the best choice for processing workflows based on the exported workflow template. The workflow engine must match all workflow requirements and the workflow should have executed successfully in previous test runs on the selected workflow engine. A list of suitable and accessible workflow engine services is presented of which the Grid workflow expert must select one item. The selection of a workflow engine implies the specification of the Grid in which the workflow will be executed. The left hand side of Figure 6.2 shows the user interface for the workflow engine selection is presented.

6.1.2 Storage selection and input file upload

The next step is the selection of a storage. Either the user selects a storage service from a list containing storage services available in the same Grid as the selected workflow engine (Figure 6.2, right hand side) or the storage will be chosen automatically. All local input files on the Grid workflow expert's hard disk will be uploaded to the selected storage.

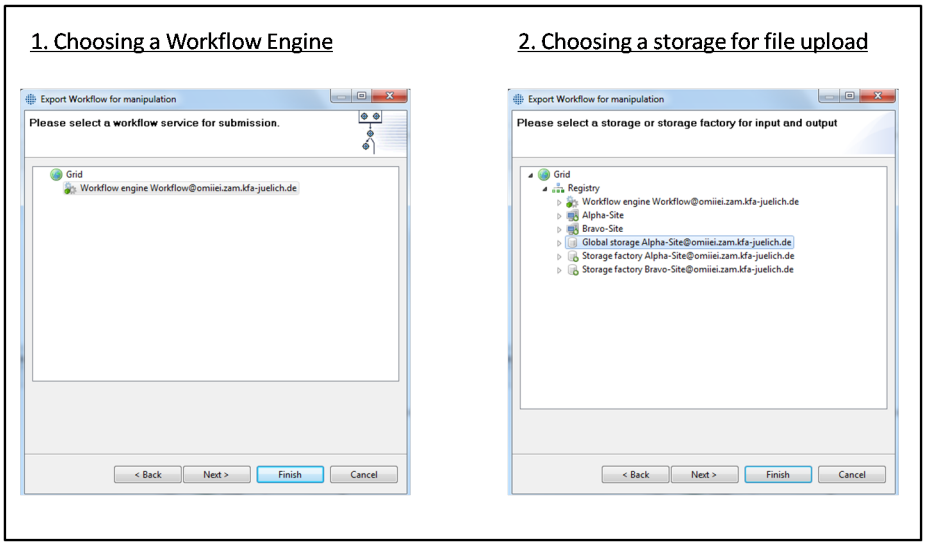


Figure 6.2: Workflow engine and storage service selection

6.1.3 Template parameter selection

The next step is the selection of template parameters, which will be substituted with placeholders in the workflow template. The process of substituting template parameter values is described in this subsection.
The explanation refers to the example workflow (1) in Figure 6.3. The workflow is converted into a template object model (2) which is used to present a choice of possible template parameters (3). The three For-Each loop parameters (iteration variable, variable modifier and terminal condition) as well as some of the Script GridBean activity input parameters, namely the primitive job parameter *ARGUMENTS*, the *SOURCE* file and the *INPUT* file set, are selected as template parameters.

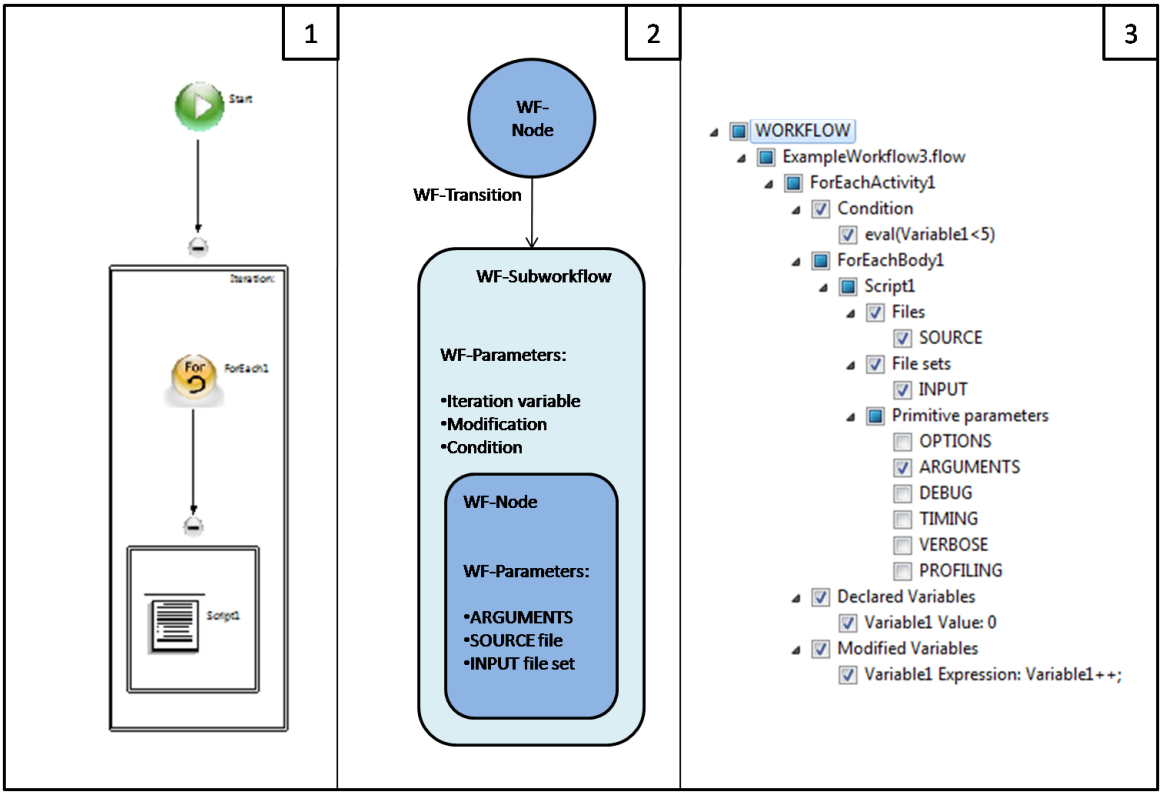


Figure 6.3: Example workflow with template parameter selection

For-Each loop variable set

The For-Each loop in the example workflow in Figure 6.3 has a variable set for the declaration of the iteration variable *Variable2*, the variable modifier and the condition, respectively. The corresponding XML workflow template extract is displayed in Listing 6.1. In order to identify the template parameter, the placeholder contains the unique loop ID and the parameter type. For example the placeholder for the terminal condition is called $$ForEachActivity1$$_$$Condition$$. Variables, modifiers and conditions of other structures are replaced similarly.

Listing 6.1: For-Each loop variable set

```
<sim:VariableSet>
            <sim:VariableName>Variable2</sim:VariableName>
            <sim:Type>INTEGER</sim:Type>
            <sim:StartValue>$$ForEachActivity1$$_$$Declaration$$</sim:StartValue>
            <sim:Expression>$$ForEachActivity1$$_$$Modification$$</sim:Expression>
            <sim:EndCondition>$$ForEachActivity1$$_$$Condition$$</sim:EndCondition>
</sim:VariableSet>
```

Primitive job parameters

UNICORE jobs are described in JSDL. Listing 6.2 shows a part of the JSDL for the Script GridBean activity. In this example, the application to be executed by the target system is Bash shell.

The Portable Operating System Interface (POSIX) application contains all primitive job parameters like *ARGUMENTS* integrated in *Environment* elements. The placeholder for primitive job parameters contains the unique activity ID (*Script1*) and the parameter name (*ARGUMENTS*).

Listing 6.2: Primitive job parameter of the Script GridBean activity

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="DEBUG"/>
      ...
      <jsdl1:Environment name="ARGUMENTS">$$Script1$$_$$ARGUMENTS$$</jsdl1:Environment>
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  ...
</sim:Activity>
```

Input files

GridBean activities can involve input files, which may be substituted by the end-user. Besides *Environment* elements, the JSDL document may contain *DataStaging* elements. A *DataStaging* element is composed of a *FileName*, a *CreationFlag* and a *Source* element including the corresponding file URI. If the end-user should be able to substitute the input file, the URI is set to the placeholder including the unique activity ID (*Script1*) and the name of the *Environment* element (*SOURCE*). This is shown in Listing 6.3.

Listing 6.3: Input file of the Script GridBean activity

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="SOURCE">input</jsdl1:Environment>
      ...
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  <jsdl:DataStaging>
    <jsdl:FileName>input</jsdl:FileName>
    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
    <jsdl:Source>
      <jsdl:URI>$$Script1$$_$$SOURCE$$</jsdl:URI>
    </jsdl:Source>
  </jsdl:DataStaging>
  ...
</sim:Activity>
```

Input file sets

Furthermore, GridBean activities and For-Each loops can contain input file sets. Input file sets are divided in two parts. An *Environment* element consists of a list with the file names of all input files in the file set. The second part is given by a list of *DataStaging* elements. Every *DataStaging* element represents one file in the file set. Listing 6.4 shows the declaration of an input file set called *INPUT*. The file name list contains two entries: *input* and $$Script1$$_$$INPUT$$. Note that the *input* entry corresponds to the value of the *FileName* element inside the *DataStaging* element. The $$Script1$$_$$INPUT$$ placeholder is used to append file names to the *Environment* element value as new files are added to the file set (which also leads to the addition of further *DataStaging* elements).

Listing 6.4: Input file set of the Script GridBean activity

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="INPUT">input $$Script1$$_$$INPUT$$</jsdl1:Environment>
      ...
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  <jsdl:DataStaging>
    <jsdl:FileName>input</jsdl:FileName>
    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
    <jsdl:Source>
      <jsdl:URI>c9m:123456/script1/input.txt</jsdl:URI>
    </jsdl:Source>
  </jsdl:DataStaging>
  ...
</sim:Activity>
```

6.1.4 Archive creation

The creation of the workflow template archive is subject to conventions, the first being that the archive name must correspond to the file names it contains. If the Grid workflow expert stores an archive, called *testworkflow.jar*, it must contain the following files:

- testworkflow.flow (XML workflow template)
- testworkflow.xml (template object model)
- testworkflow.groovy
- testworkflow.html
- manifest file

As explained earlier, Wicket has components which are rendered by HTML snippets. For each container component, such as a panel, a separate HTML snippet with the same file name as the component Java class is required. The workflow structure and the related parameter form is visualised in such a container component implemented in Groovy.

Groovy description
The layout of the the pluggable template panel component (1) is presented in Figure 6.4. The first element is a headline including the name of the workflow (2). It is modelled by a Wicket label component.
The pluggable template panel component contains a tree component for the workflow structure (3). The tree elements are links which provide a list of all related parameters in a form component embedded in a second panel component (4). The parameter list consists of selected template parameters and unselected parameters, which are visible but disabled. Three buttons are provided for submitting a workflow, for aborting the submitted workflow, as well as for resetting template parameter values (5). The last Wicket component is a text area which provides feedback messages for successful submission or error messages (6). All components are arranged in a form component (7).

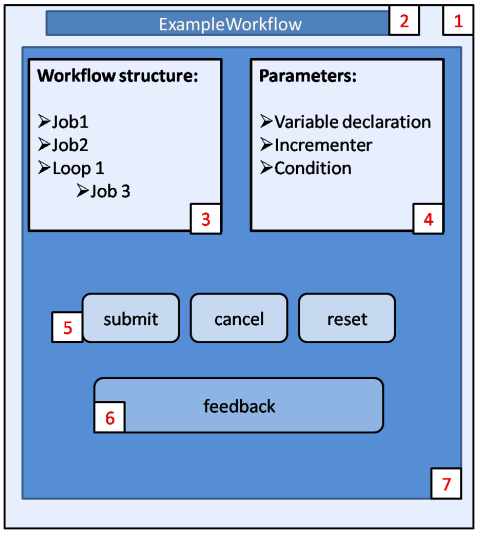


Figure 6.4: Layout of the pluggable template panel

HTML snippet

The hierarchical structure of the pluggable template panel matches the hierarchical HTML snippet. The composition must look as follows:

- Pluggable template panel
  - Label with headline (id="message")
  - form (id="form")
    - Workflow structure tree (id="tree")
    - Parameter form embedded in a panel (id="panel")
    - Buttons for resetting, workflow submitting and cancelling (id="submit/cancel/reset")
    - Text area for feedback messages (id="area")

Listing 6.5 shows the HTML snippet. Tags containing a Wicket attribute, such as id, are represented by Wicket components (compare to the list above). The remaining tags are responsible for general layouting.

Listing 6.5: HTML snippet

```
<html>
<body>
<wicket:panel>
  <h2 wicket:id="message">text goes here</h2>
  <form wicket:id="form">
    <table class="table">
      <tr>
        <td>
          <div wicket:id="tree" ></div>
        </td>
      </tr>
    </table>
    <div wicket:id="panel"></div>
    <input wicket:id="submit" type="submit" value="submit" />
    <input wicket:id="cancel" type="submit" value="cancel" />
    <input wicket:id="reset" type="submit" value="reset" />
    <textarea wicket:id="area" rows="3" cols="50"></textarea>
  </form>
</wicket:panel>
</body>
</html>
```

6.2 Web application

The following sections explain the main functions provided by the Web application: uploading a workflow template archive, manipulating a workflow template, submitting a finalised workflow template, monitoring and fetching output from a submitted workflow by using the example workflow template archive called *testworkflow* from the previous section.

6.2.1 Uploading a workflow template archive

The file upload component is displayed in Figure 6.5 (1). It contains a Wicket upload field for files and a Wicket button component which triggers the upload of the chosen file. After uploading the archive, a new entry in the workflow template repository will be produced which is visualised by a new link. For the example workflow *testworkflow* this is shown in Figure 6.5 (2). The user can delete the entry by selecting it and using the delete button. The link leads to parsing and loading the related Groovy component, which embeds the template panel. The Groovy component is then displayed on the right hand side in the workflow template management Web page (3).

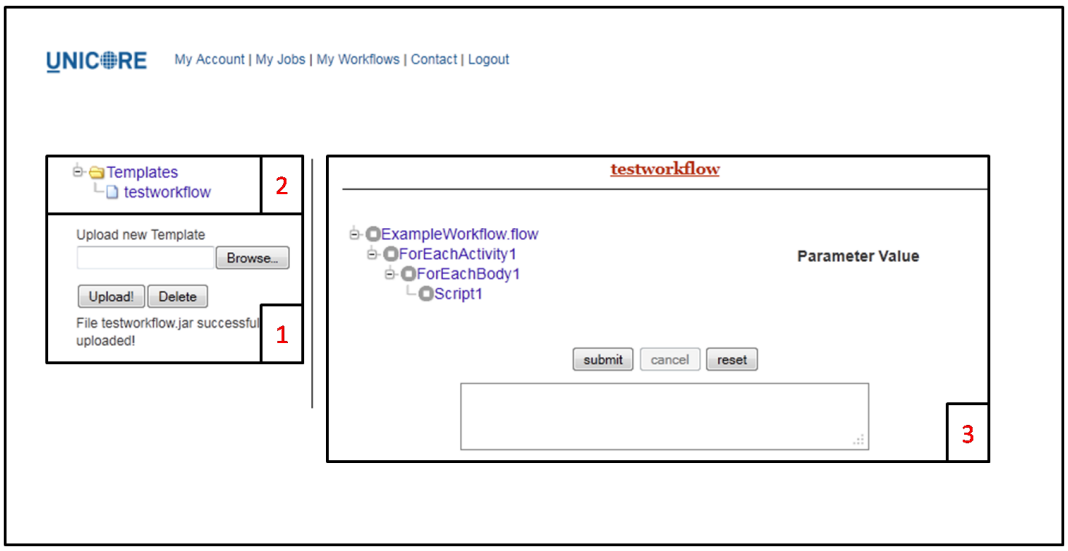


Figure 6.5: Web page illustration of an uploaded workflow template archive

In addition to the link creation, the uploaded archive is stored in the end-user's directory at the Web application server. Furthermore a directory with the same name as the archive is created there. Inside this directory, all important files like the Groovy and HTML files are placed. Wicket needs the HTML snippet for rendering the Groovy component.

In order to store submitted workflows, the related template object models are copied to a new subdirectory with the name of the submitted workflow. For the example *testworkflow* the directory structure is organised as presented in Figure 6.6.

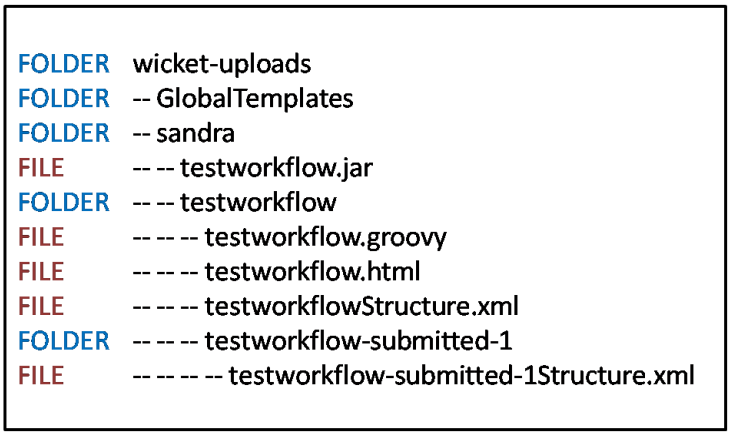
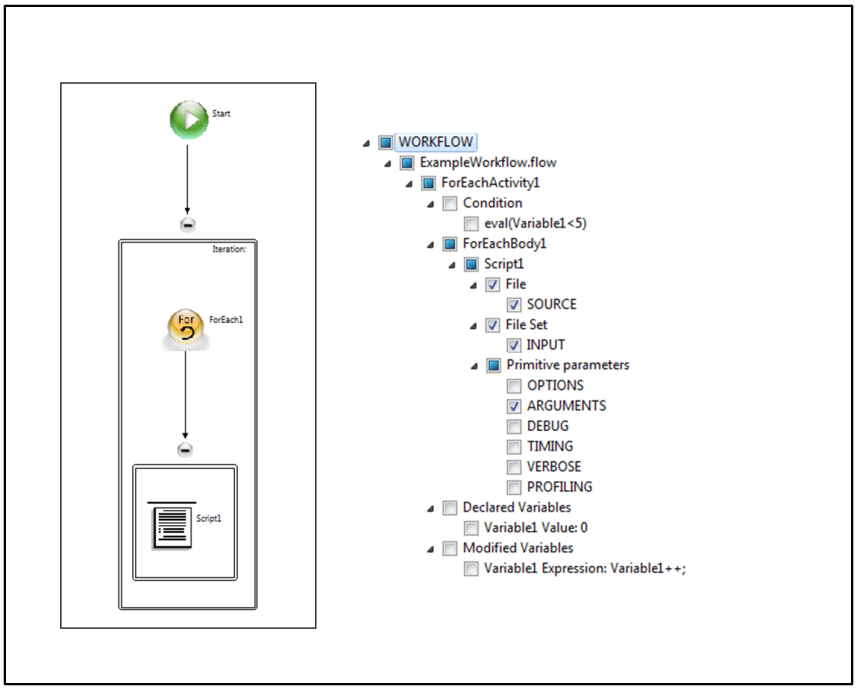


Figure 6.6: File system organisation

If the user decides to delete a template, the template directory and all subdirectories and files will be deleted as well as the link. The user can also delete submitted workflows without deleting the template.

6.2.2 Manipulating a workflow template

The parameter form embedded in the template panel allows the substitution of template parameter values. There are six different workflow parameter types which are divided into two different groups. Workflow variables, modifiers, conditions and primitive job parameters are modelled as textual values. Files and file sets belong to file values. Figure 6.7 shows the relation between the created workflow in the URC workflow editor, the export presentation and the visualisation of the workflow structure with the related parameter form in the Web application. The illustrated parameter form allows the substitution of all template parameter values. The workflow structure is visualised in a Wicket tree component. Clicking a tree entry presents the parameter form for the corresponding workflow part. For example a click on *Script1* produces a presentation of all primitive job parameters as well as files and file sets defined for *Script1*. All selected template parameters are presented by editable components. In contrast, unselected parameters are presented by disabled components merely displaying the defined value.



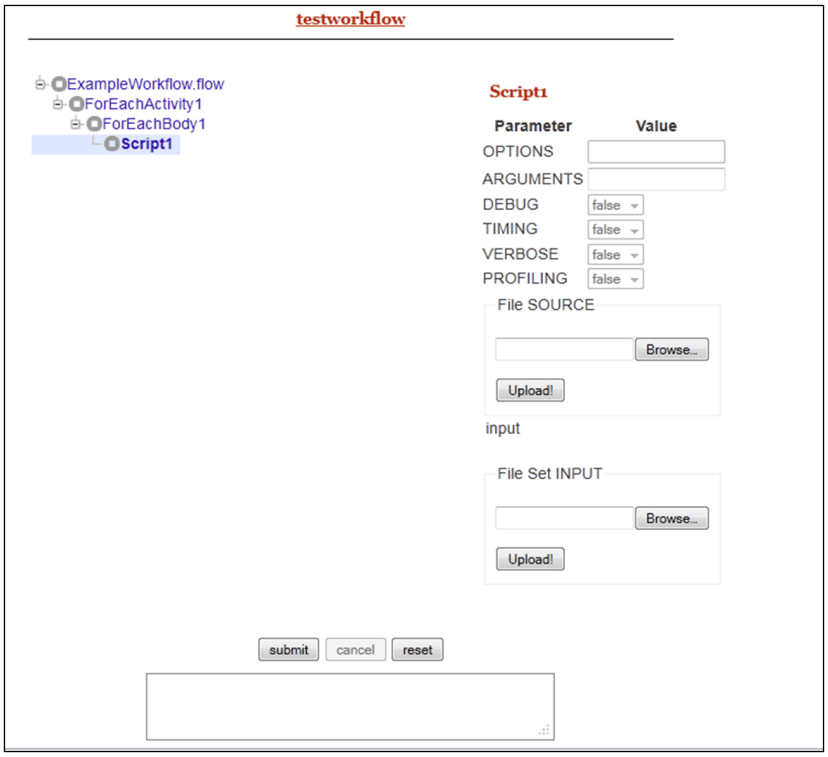


Figure 6.7: Relation between the presentation of workflows in the URC workflow editor and in the Web application

Text values

The presentation of text values depends on the parameter type. Floating point, integer, string and Boolean parameters are possible. Parameters of one of the first three types are presented in a text field component, shown in Figure 6.8.

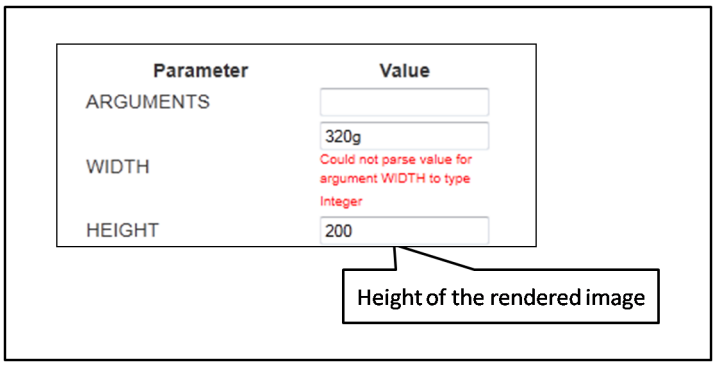


Figure 6.8: Wicket text field component

As a default the original parameter value is displayed. Changing this value leads directly to update the template object model. This is handled by the behaviour of the text field component. The label to the left of the text field component displays the variable name. Parameters with the Boolean type are presented via a combo box. Available values are *false* and *true*. This is illustrated in 6.9.

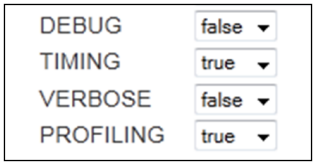


Figure 6.9: Wicket combo box component

The user input is validated using meta information such as the parameter type. To this end a specialised Wicket validator has been developed. Further meta data are the parameter descriptions, which are presented in tool tip components.

File values

File values are more complex and visualised by a file upload component, shown in Figure 6.10. The file upload component consists of an upload field for files and a button component which triggers the upload of the chosen file, similar to the upload component for the workflow template archive.
The legend of the file upload component contains the type (file or file set). A tool tip component for the file upload field contains the file description and the actual file name, or, in case of file sets, a list of included file names.
If a user chooses a new file, it will be directly uploaded to the workflow working directory which is deployed next to the Web application. The address of a file on the storage, given as a URI, can be used to transfer this file from or to other storages.

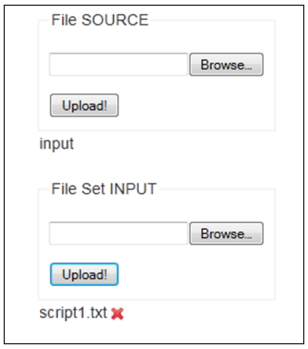


Figure 6.10: File upload component

6.2.3 Submitting a workflow

Each pluggable template panel provides a submit button which allows the submission of the finalised workflow (Figure 6.11, (1)). It also provides a user notification text area component (2), described in the section *Monitoring a submitted workflow*.
The submit operation consists of two steps. First, all placeholders in the XML workflow template are substituted by the new values that have been set in the template object model. Next, the workflow description is sent to the workflow engine chosen during the workflow template export, where the workflow will be processed.

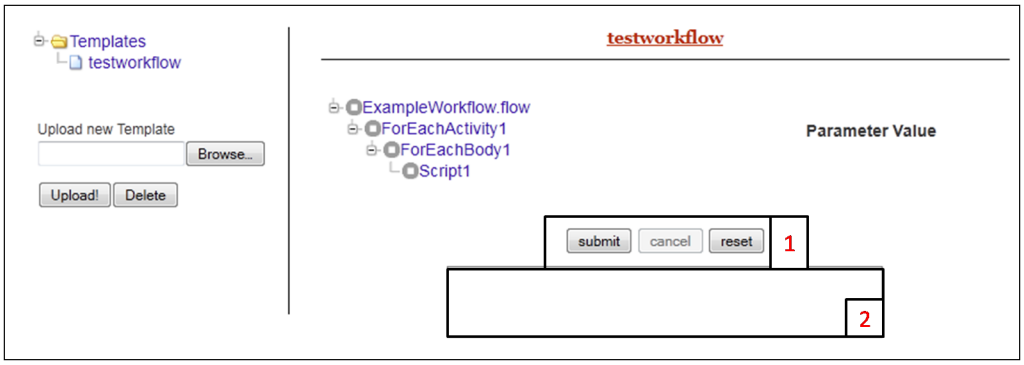


Figure 6.11: Interface for the submit function

For each template parameter value, the according placeholder in the workflow description needs to be substituted with the new value or reset to the default value. In the example, the three parameters *ARGUMENTS*, *SOURCE* file and *INPUT* file set, which were marked as template parameters during the export function (Section 6, Template parameter selection), are used to demonstrate the substitution.

Primitive job parameters
For primitive parameters, the placeholder contains the activity ID and the parameter name. For the job parameter *ARGUMENTS* the placeholder $$Script1$$_$$ARGUMENTS$$ (Listing 6.6) is substituted with the new value *–info* displayed in Listing 6.7. For workflow variables, modifiers and conditions the processing is similar.

Listing 6.6: Placeholder for primitive job parameter

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="DEBUG"/>
      ...
      <jsdl1:Environment name="ARGUMENTS">$$Script1$$_$$ARGUMENTS$$</jsdl1:Environment>
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  ...
</sim:Activity>
```

Listing 6.7: Substitution of placeholder for primitive job parameter

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="DEBUG"/>
      ...
      <jsdl1:Environment name="ARGUMENTS">--info</jsdl1:Environment>
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  ...
</sim:Activity>
```

Input files

New files are stored on a UNICORE storage service as described in the previous subsection. In order to substitute the corresponding placeholder in the XML workflow template, the URI of the file is needed.

Listing 6.8: File URI example

```
https://zam079.zam.kfa-juelich.de:6000/DEMO-SITE/services/StorageManagement?
res=4cb466ac-8707-46b2-b501-53f3ca241e50#testworkflow/script1/file1.txt
```

The URI consists of the storage address (*https://zam079.zam.kfa-juelich.de:6000/DEMO-SITE/services/StorageManagement*), the reference to a concrete storage management service instance (*?res=4cb466ac-8707-46b2-b501-53f3ca241e50#*) and the relative path to the file (*testworkflow/script1/file1.txt*). In order to substitute a file parameter value, the URI inside the *DataStaging* element must be replaced, as shown in Listing 6.9 and Listing 6.10. A substitution of the file name (input) is not necessary. The Baseline File Transfer scheme (BFT) prepended to the file URI specifies the file transfer protocol to be used.

Listing 6.9: Placeholder for input file

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="SOURCE">input</jsdl1:Environment>
      ...
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  <jsdl:DataStaging>
    <jsdl:FileName>input</jsdl:FileName>
    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
    <jsdl:Source>
      <jsdl:URI>$$Script1$$_$$SOURCE$$</jsdl:URI>
    </jsdl:Source>
  </jsdl:DataStaging>
  ...
</sim:Activity>
```

Listing 6.10: Substitution of placeholder for input file

```
<sim:Activity Id="Script1" Type="JSDL" Name="JSDL">
  ...
  <jsdl:Application>
    <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
      <jsdl1:Environment name="SOURCE">input</jsdl1:Environment>
      ...
    </jsdl1:POSIXApplication>
  </jsdl:Application>
  <jsdl:DataStaging>
    <jsdl:FileName>input</jsdl:FileName>
    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
    <jsdl:Source>
      <jsdl:URI>BFT:https://zam079.zam.kfa-juelich.de:6000/DEMO-SITE/services/StorageManagement?
res=4cb466ac-8707-46b2-b501-53f3ca241e50#testworkflow/script1/file1.txt</jsdl:URI>
    </jsdl:Source>
  </jsdl:DataStaging>
  ...
</sim:Activity>
```

Input file sets

File sets can be extended by new file values, and files from the file set can be deleted. The file values are stored in the template object model. An example file set *INPUT* for *Script1* includes the placeholder $$Script1$$_$$INPUT$$. For each new file a new *DataStaging* element must be inserted and the *Environment* element value must be extended with the file name. Files which were pre-defined in the file set can also be deleted. In this case the file name must be deleted from the list and the *DataStaging* element needs to be deleted from the job description.
For file sets, which are declared in For-Each loops, the processing is similar. The URI is split into the base directory and the file name.

Output file addresses

Finally, all output file addresses in the XML workflow template must be adjusted. In order to prevent address clashes, output file addresses contain a placeholder for the unique workflow ID which is determined during workflow submission (see Listing 6.11). This placeholder is substituted shortly before the workflow is submitted (see Listing 6.12).

Listing 6.11: Placeholder for output file address

```
<jsdl:DataStaging>
  <jsdl:FileName>stdout</jsdl:FileName>
  <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
  <jsdl:Target>
    <jsdl:URI>c9m:${WORKFLOW_ID}/Script1:::${CURRENT_TOTAL_ITERATOR}/stdout</jsdl:URI>
  </jsdl:Target>
</jsdl:DataStaging>
```

Listing 6.12: Substitution of placeholder for output file

```
<jsdl:DataStaging>
  <jsdl:FileName>stdout</jsdl:FileName>
  <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
  <jsdl:Target>
    <jsdl:URI>c9m:ecb096a3-a465-4f07-973e-aaa84250d857/Script1:::${CURRENT_TOTAL_ITERATOR}/stdout</jsdl:URI>
  </jsdl:Target>
</jsdl:DataStaging>
```

6.2.4 Monitoring a submitted workflow

During workflow processing the execution state can be monitored. In order to refresh the monitoring view regularly, Ajax is employed. In the background the Web application sends requests to the UNICORE workflow system in order to retrieve the workflow state. The workflow structure is displayed in a tree view. A node is a workflow part, such as an activity or a loop structure, which has an execution state. Figure 6.12 visualises all possible states for the workflow parts. The execution states have the following meanings (from left to right): A workflow part which has not been processed yet has the state *undefined*. The state *running* is set for a workflow part that is processed at this moment. There are three terminal states for a workflow part. The first one indicates a *successful* execution. The next icon indicates a *failed* execution (visualised by the fourth icon). A failure may have different reasons, e.g. that no suitable computational resource could be found or a file transfer failed. The last icon symbolises the *abort* state which is set if a user has cancelled the workflow execution.

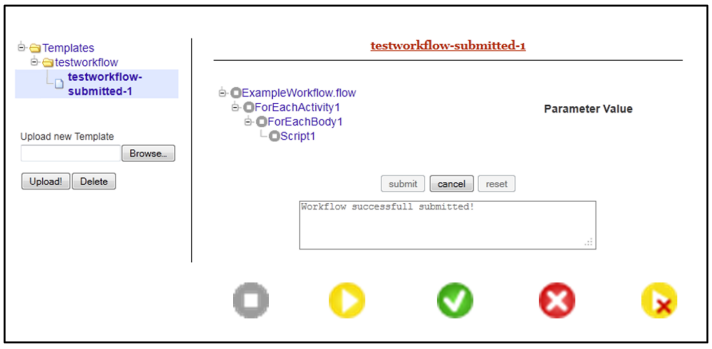


Figure 6.12: Workflow states

6.2.5 Fetching output from a submitted workflow

During workflow execution, output files are created and stored back to the workflow working directory in order to be accessible via the Web application in a persistent manner.
All created files, including input and output files, are visualised by new nodes beneath the corresponding workflow part in the monitoring tree view. If a workflow part is executed multiple times in a loop, a new node labelled with the iteration name will be created as a parent node for nested workflow parts or files created during this iteration. Figure 6.13 shows a list of files beneath the tree node labelled *Script1_iteration_4*.
All tree nodes are links which can be selected to show a detailed view of file meta information and to allow the user to download files. The behaviour of a *Download* link involves opening a download pop-up window which asks the user whether to save or open the file. Choosing *save* allows the user to download the file to the local disk. Opening a file involves selecting and executing an installed program on the local system. This is presented in Figure 6.13. In this case the user chose the file *stdout* for downloading to their local disk.

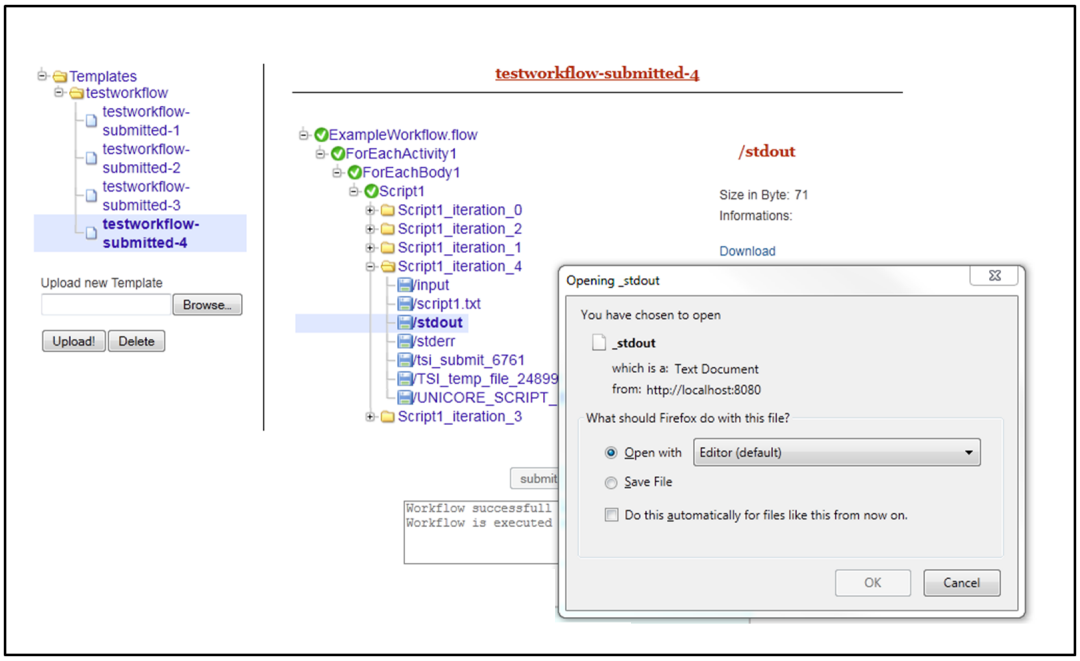


Figure 6.13: Downloading output files

Chapter 7

Example scenario

The previous chapters explained the design and implementation of the export function and the Web application. In this chapter, in order to illustrate the process, they will be applied step by step to an example workflow. The screenshots are based on the URC version 6.4.0 and the Web browser Firefox 4.0.1. At first, the Grid workflow expert creates a workflow in the graphical workflow editor provided by the URC.

7.1 Workflow creation

The example workflow assembles a video file combining multiple image files from the Blender application, which is a free 3D modelling and rendering software. The workflow is presented in Figure 7.1.
At first, workflow variables defining the number of frames as well as variable modifiers, which change the variable values, are needed. The first step of the workflow is the declaration of a workflow variable, called *Variable1*, with an initial value of zero. The variable type is declared as integer. This variable is needed for the definition of the number of image files which will be generated by each Blender application run. The next part is a loop structure, specifically a For-Each loop. The iteration variable is called *Variable2* of type integer and the initial value zero. The modifier expression *Variable2=Variable2+10* means that the value of *Variable2* will be raised by ten in each step. This will be repeated until the variable reaches a value of 80. The For-Each body contains two activities, a Variable Modifier activity and a Blender GridBean activity. The Variable Modifier activity is used to set the value of *Variable1* to *Variable2+9*. The parameters for the Blender GridBean activity are visualised in Figure 7.2. Besides the job name and the format, it is possible to choose between rendering an animation or rendering single frames. In this case the animation option is chosen. The Blender scene file, which is needed for the rendering process, is defined as *input.blend*. The start and end variables, which are defined by *Variable2* and *Variable1*, are used to define the number of frames for the animation rendering. A look at the explained modifier expressions shows that in every loop iteration exactly ten frames will be created, in the first iteration from 0 to 9, then from 10 to 19 etc., until the start value reaches 80. The For-Each loop is followed by a Script GridBean activity, which is presented in Figure 7.3.

The Script activity is used to generate a video from the output files of the Blender GridBean activity. Its input files are the generated output files of the Blender GridBean activity, and the script code contains the application call *ffmpeg* with specified arguments. The *video.wmv* file represents the generated video.

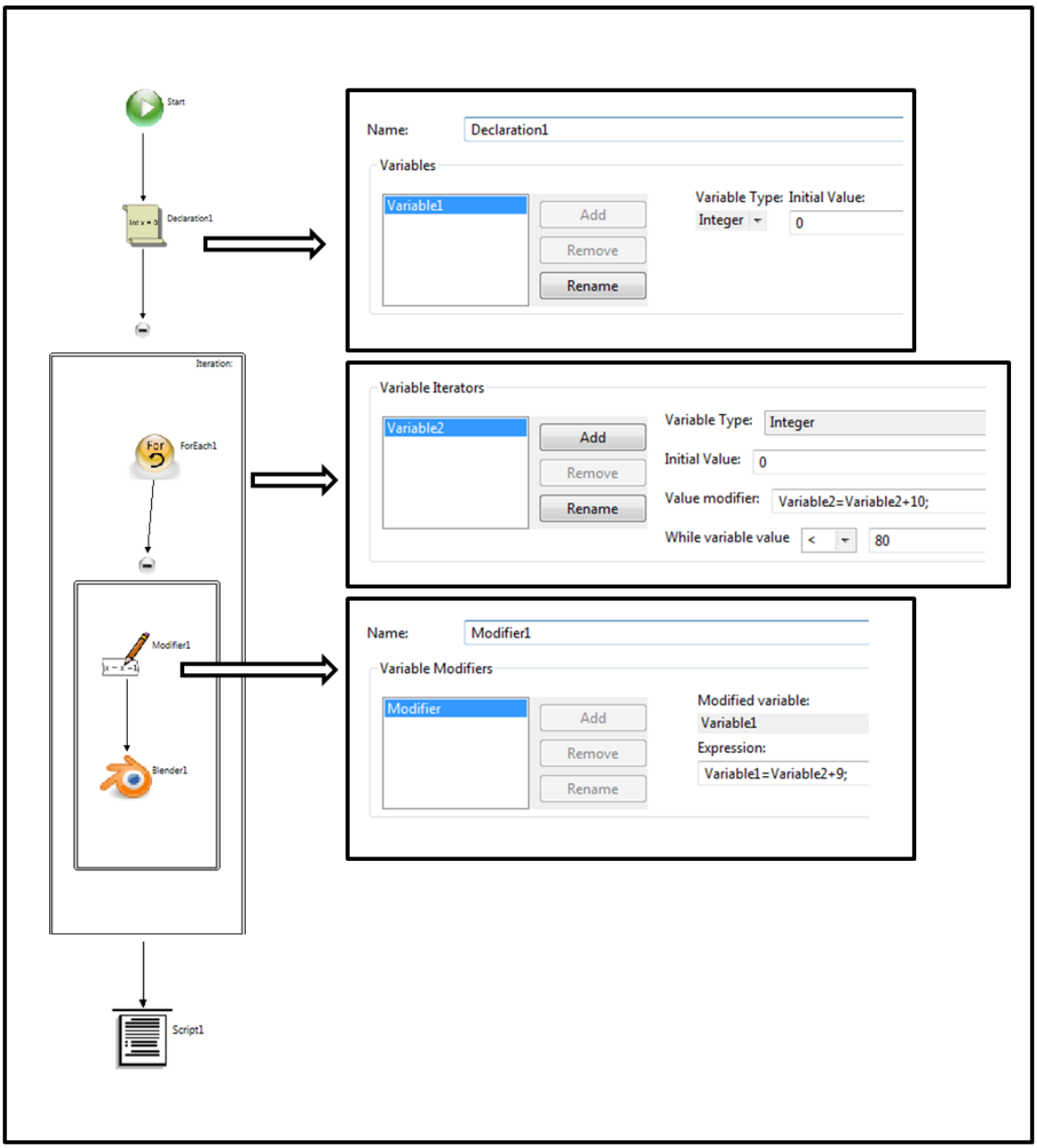


Figure 7.1: Workflow construction

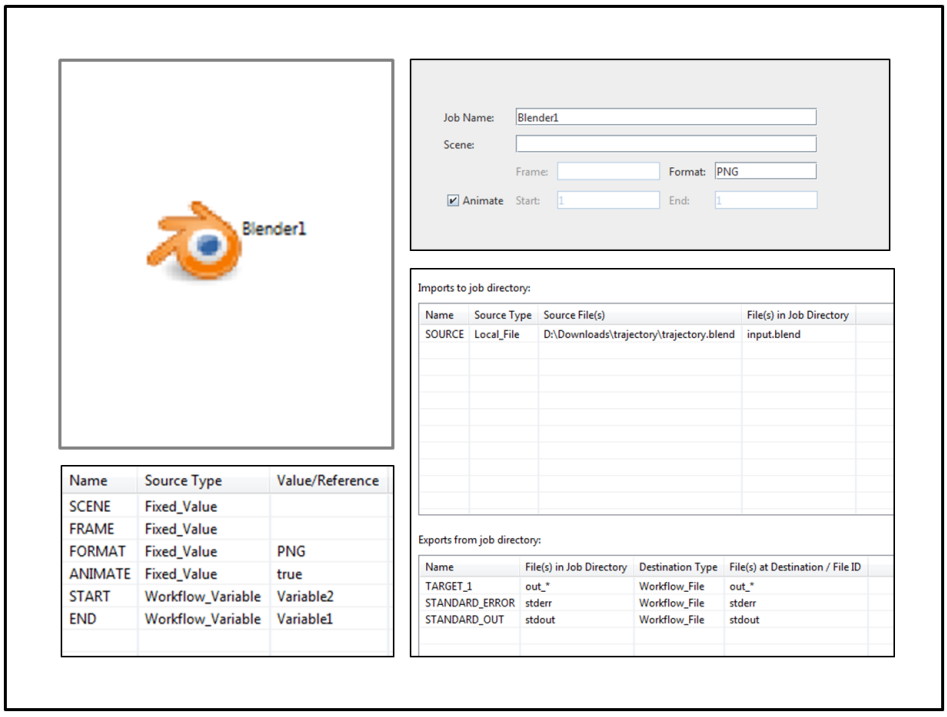


Figure 7.2: Blender GridBean activity with preferences

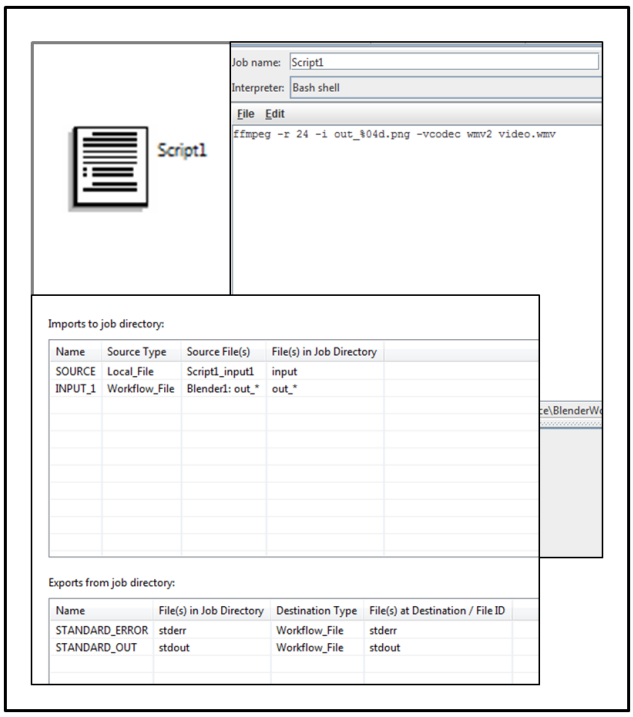


Figure 7.3: Script GridBean activity with preferences

7.2 Workflow template export

The following subsections explain the export steps beginning with the selection of the workflow engine and the storage service.

7.2.1 Workflow engine and storage service selection

The first step is to choose an accessible workflow engine. Figure 7.4 shows the selection. All workflows generated by this workflow template will be submitted to the selected workflow engine.
Next, a storage service can be selected. The list of available storage services depends on the chosen Grid. The Grid workflow expert chooses a storage service where all the local input files (the Blender scene and the script file) are uploaded to.

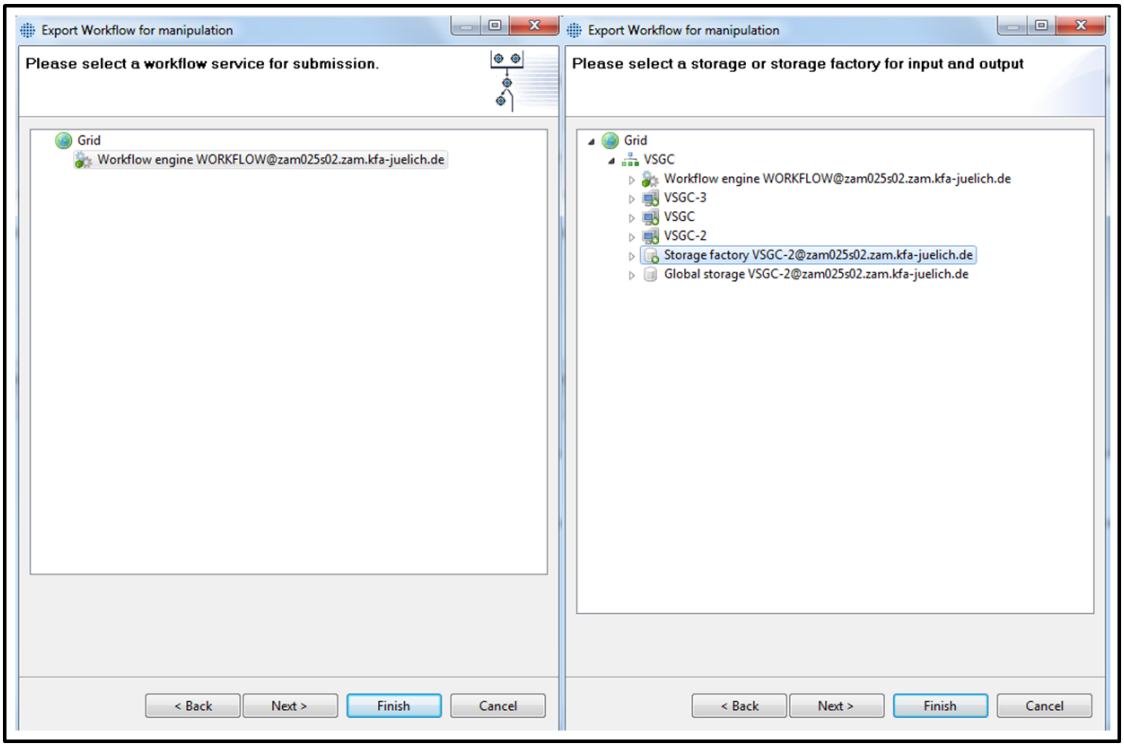


Figure 7.4: Choosing the workflow engine and the storage service

7.2.2 Template parameter selection

The presentation of template parameters is shown in Figure 7.5. The Grid workflow expert selects the following parameters: the Blender scene file and the condition expression of the For-Each loop.

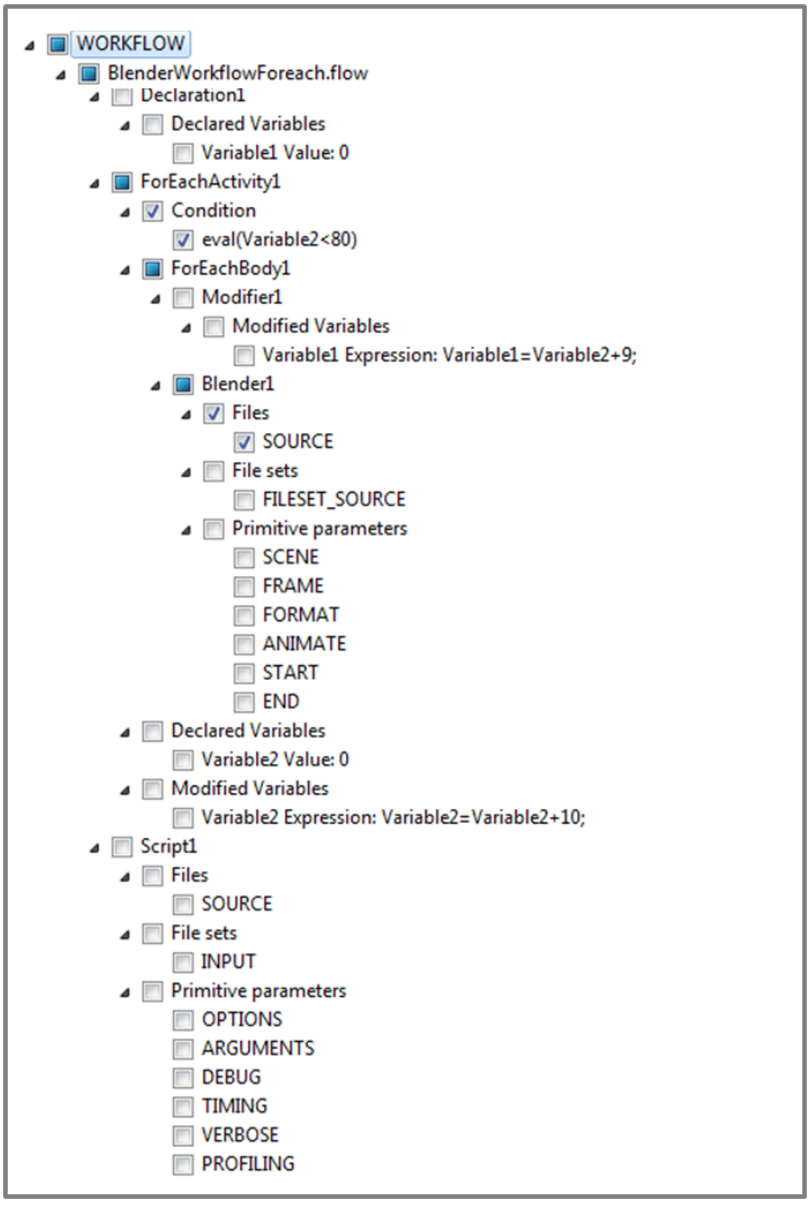


Figure 7.5: Selection of template parameters

These template parameters are marked in the XML workflow template.
This is demonstrated for the Blender scene file and for the condition defined in the For-Each variable set in Listing 7.1.

Listing 7.1: Placeholders in the Blender XML workflow template

```
<jsdl:DataStaging>
  <jsdl:FileName>input.blend</jsdl:FileName>
  <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
  <jsdl:Source>
    <jsdl:URI>$$Blender1$$_$$SOURCE$$</jsdl:URI>
  </jsdl:Source>
</jsdl:DataStaging>

<sim:VariableSet>
  <sim:VariableName>Variable2</sim:VariableName>
  <sim:Type>INTEGER</sim:Type>
  <sim:StartValue>0</sim:StartValue>
  <sim:Expression>Variable2=Variable2+10;</sim:Expression>
  <sim:EndCondition>$$ForEachActivity1$$_$$Condition$$</sim:EndCondition>
</sim:VariableSet>
```

7.2.3 Archive creation

For quick access to template parameters, the workflow is converted into a template object model which is presented in Figure 7.6.

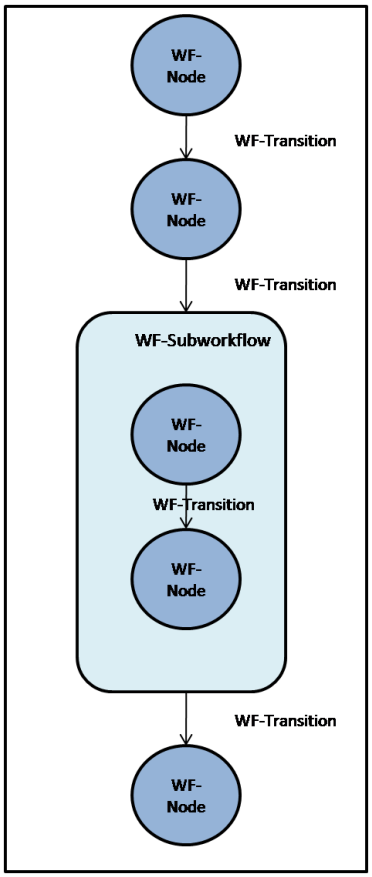


Figure 7.6: Template object model of the Blender workflow

The Grid workflow expert stores the created archive with the name *blender.jar*. Note that this name also specifies the names of the included files, see Figure 7.7. Now the end-user is able to upload the created jar file to the Web application.

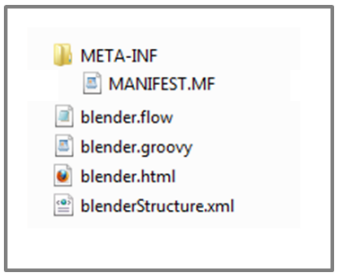


Figure 7.7: Context of the *blender.jar* file

7.3 Web application

After uploading the archive, the end-user changes both template parameter values. Then the user submits the generated workflow. By monitoring the submitted workflow, the user tracks the workflow's state. At last the user fetches the generated video file.

7.3.1 Uploading the archive

The file upload component allows a user to select a file and upload it to the Web application. The user chooses the previously stored *blender.jar* file and then presses the upload button to trigger the operation. If a problem is detected during the upload process the user will be informed. In the same way they get a message when the process has successfully finished as in this case (see Figure 7.8).

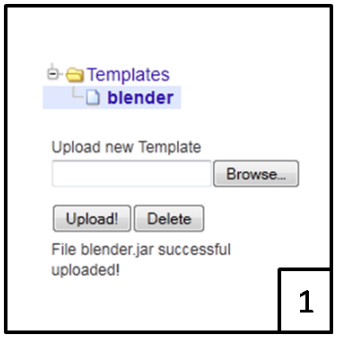


Figure 7.8: Upload the workflow template archive *blender.jar*

7.3.2 Substituting template parameter values

The controls for the two selected template parameters are enabled for modification (Figure 7.9). The component for the Blender scene file is a file upload field (1). The condition of the For-Each loop is visualised in a text field component with a corresponding label (2). The user changes both parameter values. The Blender scene file is changed to another file called *unicore_demo.blend*, and the limit for the counter for the For-Each loop is changed from 80 to 20 which leads to rendering 20 frames instead of 80.

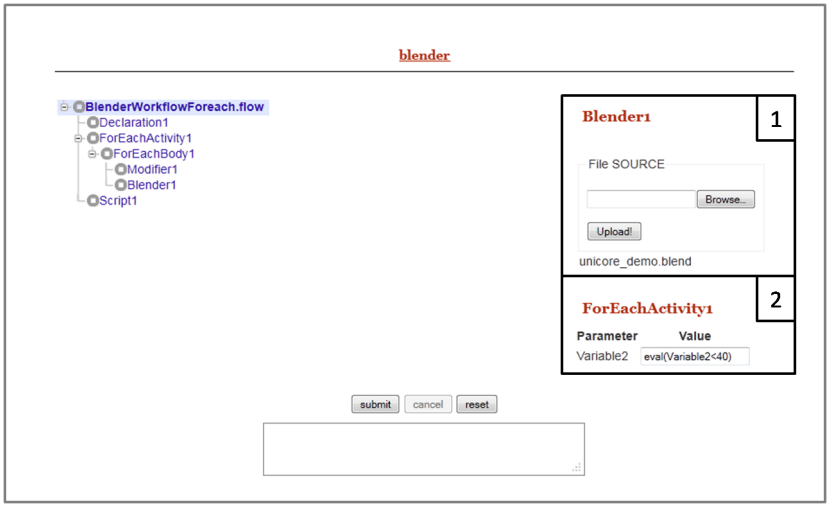


Figure 7.9: Template panel for the Blender workflow template

The corresponding elements in the XML workflow template are presented in Listing 7.2 below.

Listing 7.2: Substitution of placeholders in the XML workflow template

```
<jsdl:DataStaging>
  <jsdl:FileName>input.blend</jsdl:FileName>
  <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
  <jsdl:Source>
    <jsdl:URI>BFT:https://zam079.zam.kfa-juelich.de:6000/DEMO-SITE/services/StorageManagement?
res=4cb466ac-8707-46b2-b501-53f3ca241e50#blenderworkflowforeach/blender1/unicore_demo.blend</jsdl:URI>
  </jsdl:Source>
</jsdl:DataStaging>

<sim:VariableSet>
  <sim:VariableName>Variable2</sim:VariableName>
  <sim:Type>INTEGER</sim:Type>
  <sim:StartValue>0</sim:StartValue>
  <sim:Expression>Variable2=Variable2+10;</sim:Expression>
  <sim:EndCondition>eval(Variable2<20)</sim:EndCondition>
</sim:VariableSet>
```

7.3.3 Submission and monitoring

Now the user can submit the created workflow to the workflow engine. If the submit process is successfully done the user gets a notification. The user interface for monitoring is shown in Figure 7.10.

Every few seconds an automatic update operation looks for the element status and visualises it via different icons. The update operation also looks for newly generated files and visualises them.

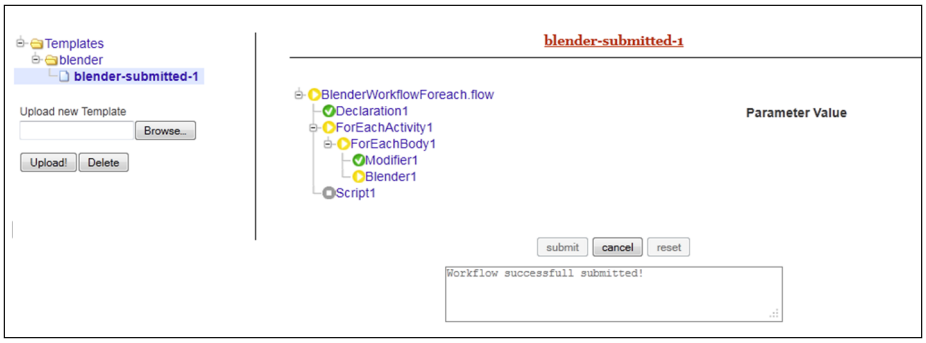


Figure 7.10: Monitoring the workflow

7.3.4 Fetching output

The user can download generated output files via the Web application. The result of the example workflow is a video called *video.wmv*. As shown in Figure 7.11, the file has a size of 99,789 Bytes (1) and the user opens it on their local system with a suitable program, for example the Windows Media Player (2).

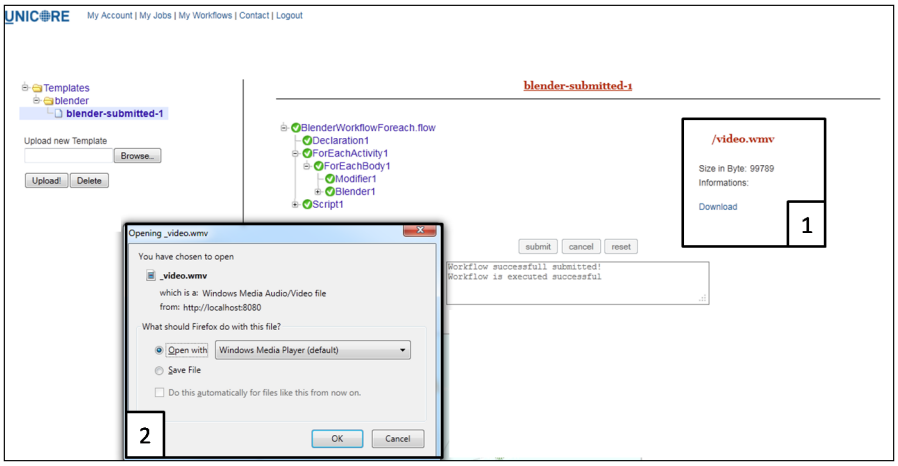


Figure 7.11: Fetching the resulting video file

Chapter 8

Conclusion and further developments

8.1 Conclusion

This Master thesis comprises the development of an export function, which generates workflow template archives, and the integration of workflow templates in the Web Application based on the current UNICORE Web client.
The Grid workflow expert has the experience to create workflows and use complex functions provided by the URC. Furthermore, the expert generates workflow templates based on created workflows using the new export function. These templates are uploaded to the Web application by end-users who change template parameter values and thereby create concrete workflows for each scenario. These workflows are submitted to the workflow engine selected by the Grid workflow expert.
The export function allows the selection of template parameters. For the substitution of template parameters with placeholders, an analysis of all existing workflow parameters is required. The access to template parameters is handled by a template object model which is part of the workflow template archive. Pluggable template panels are used to provide a user interface for parameter forms; these panels are based on Java classes and HTML snippets developed during the course of this Master thesis. For the implementation of the Java class, the scripting language Groovy has been used in order to avoid Java compiling during the export process.
The Web application has been implemented using Apache Wicket. It provides a workflow template repository and offers operations for uploading, deleting and manipulating workflow templates. Workflow submission and monitoring are provided, as well as cancelling and fetching output of submitted workflows. In addition, a workflow working directory is required to make new files available to the UNICORE workflow system and to store output files from executed workflows. Since the UNICORE Web client is still work in progress, the newly developed workflow page will be integrated together with other new features and changes in the near future.

8.2 Further developments

User identification
Currently the user identification is based on user name and password. Behind the scenes, the portal uses a single certificate to authenticate itself to the UNICORE workflow system. This represents a weak security model because all end-users of the Web application act as the same Grid user and can access each other's resources. In the future, this will change as the user database of the UNICORE Web client framework will become fully functional, allowing the Web application to act on behalf of the users through trust delegation ([2]).

Direct template upload from URC to the Web application
At the moment users need to store workflow template archives on their local disk and then upload them to the Web application in a second step. A more comfortable solution can be achieved by extending the export function and uploading created archives directly from the URC to the Web application.

Using the JavaScript InfoVis Toolkit (JIT) for visualising workflows
The presentation of workflow templates is currently done in tree form. A more suitable presentation can be realised by applying JavaScript based technologies, e.g. JIT[1]. It is possible to create an interactive graph-based workflow presentation as presented in Figure 8.1.

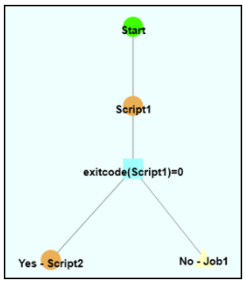


Figure 8.1: Workflow presentation with JIT

Workflow repository
The Web application is used to manage workflow templates which are stored on a per-user basis. It should be possible to share templates, so that other users could also manipulate them and submit finalised workflow templates to a workflow engine. As a result, users would also be able to quickly evaluate workflows from other users. Another feature is to create template categories (e.g. biological templates) and build a hierarchical template structure. It should be possible to sort templates and to search for templates by name and category.

[1] JavaScript InfoVis Toolkit: http://thejit.org/

Exploiting the Web application in projects like UIMA-HPC
In research projects, where workflows need to be executed many times and only few parameter values are changed in every scenario, workflow templates are useful. For instance, the UIMA-HPC project employs workflows where the input data varies for each execution, whereas the overall workflow structure remains unchanged. Under these circumstances workflow templates would allow project members to share the developed workflows without having to create them from scratch.

Glossar

Ajax	Asynchronous JavaScript and XML.	11, 12, 16
API	Application Programming Interface.	4, 6, 10
behaviour	A Wicket behaviour contains component operations for an event.	13, 14, 43
component	A Wicket component is responsible for view and controller part.	12–14, 16, 38–41, 43, 44
end-user	Users who only want to change few parameters in a workflow.	19, 20, 22, 25, 40, 54, 55
Grid middleware	Coordinates distributed resources.	I, 1, 3, 6
Grid workflow expert	The expert user who creates workflows and workflow templates.	1, 19, 24, 34, 35, 38, 49, 52–54
GridBean	Allows to define abstract jobs.	7, 28–30, 35–37, 49, 50
Groovy	Provides script code syntax for the implementation of Java classes.	I, 16, 17, 32, 38, 40
HTML	HyperText Markup Language.	9, 13, 15, 16, 32, 38, 39
model	A Wicket model is used to store data.	12–14, 16
MVC	Model View Controller design pattern.	9, 12
registry	Stores a list of all services accessible in a Grid.	5, 6
service orchestrator	Part of the UNICORE workflow system which processes jobs integrated in work assignments.	6, 7, 9
template object model	A model for the corresponding workflow template.	27, 29, 35, 38, 41, 43, 44, 46, 54
template panel	A Wicket container component which is used to display the workflow structure and template parameters.	25, 27, 31, 32, 38, 39, 44

template parameter	A selected workflow parameter which is substituted by a placeholder in the submittable workflow description.	19–21, 25, 27, 29, 34, 35, 38, 41, 44, 53–56
UNICORE	Uniform Interface to Computing Resources.	I, 1, 3–7, 22, 36
URC	UNICORE Rich Client, graphical user interface for UNICORE access.	4, 6, 34, 41, 49
Web application	Application that is accessible over a network.	9, 11–15, 20, 21, 23–25, 31, 32, 40, 41, 43, 47, 49
Web application framework	Software framework which allows the development of dynamic Websites and Web applications.	I, 9–12, 31
Wicket	Allows the implementation of a Web application.	I, III, 12–14, 16, 31, 32, 38–41
workflow	A workflow is modelled as a compound directed graph. It consists of jobs and structures.	1, 2, 6, 7, 19–25, 28–32, 34, 35, 38, 40, 41, 44, 46, 47, 49, 52, 55–57
workflow engine	Core system of the UNICORE workflow system which processes workflow descriptions.	6, 7, 27, 32, 34, 35, 44, 51, 52, 56
workflow system	Provides workflow processing functions.	1, 7, 22, 25, 31, 32, 46
workflow template	Submittable workflow description in XML with placeholders.	2, 19–22, 24, 25, 27, 28, 31, 32, 35, 38, 40, 41, 44–46, 52, 54, 56
workflow template archive	The archive comprises the template object model, the workflow template, and files for the corresponding pluggable template panels.	20, 24, 31, 32, 38, 40, 43
workflow template repository	A repository which stores workflow templates and provide access to them via interfaces.	20, 25, 31, 32, 40
workflow working directory	A storage for new uploaded input files and produced output files	21, 25, 31–33, 43, 47, 58
XML	Extensible Markup Language.	4, 10, 12

Bibliography

[1] Ali Anjomshoaa, Fred Brisard, Michel Drescher, Donal Fellows, An Ly, Stephen McGough and Darren Pulsipher. Job Submission Description Language (JSDL), Specification, Version 1.0. `http://www.gridforum.org/documents/GFD.56.pdf`, 7.11.2005.

[2] Krzysztof Benedyczak, Piotr Bala, Sven van den Berghe, Roger Menday, and Bernd Schuller. Key aspects of the UNICORE 6 security model. *Future Generation Computer Systems*, 27, 2011.

[3] Krzysztof Benedyczak, Jason Milad Daivandy, Bastian Demuth, and Bernd Schuller. Chemomentum - WP 1 Architecture and Software framework. Deliverable D1.6, 2009.

[4] UNICORE community. Guide to XACML security policies. `http://www.unicore.eu/documentation/manuals/unicore6/unicorex/policies.html`, 2010.

[5] UNICORE community. UNICORE Commandline line Client. `http://www.unicore.eu/documentation/manuals/unicore6/ucc/`, 2010.

[6] Martin Dashorst. *Wicket in Action*. Manning Publications Co., 2009. Foreward by Jonathan Locke.

[7] Guillaume Laforge Dierk Koenig, Paul King and Jon Skeet. *Groovy in Action, Second Edition*. Manning Publications Co., 2011. Foreward by James Gosling.

[8] Ralf Ebert. Eclipse Workbench, Views und Perspektiven. `http://www.ralfebert.de/rcpbuch/workbench/`, 24. Juni 2009.

[9] SELFHTML e.V. Stylesheets (CSS). `http://de.selfhtml.org/css/`, 2007.

[10] I. Foster, A. Grimshaw, P. Lane, W. Lee, M. Morgan, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. OGSA Basic Execution Service Version 1.0, OGF Specification GFD. `http://www.ogf.org/documents/GFD.108.pdf`, August 2007.

[11] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, 1998. Eds.

[12] Jesse James Garrett. Ajax A new Approach to Web Applications. `http://adaptivepath.com/ideas/essays/archives/000385.php`, 2005.

[13] Mark Johnson Inderjeet Singh, Beth Stearns and Enterprise Team. Web-Tier Application Framework Design. `http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html`, 2002.

[14] Ahmed S. Memon, Mohammad S. Memon, Philipp Wieder, and Bernd Schuller. CIS: An Information Service Based on the Common Information Model. http://www.computer.org/portal/web/csdl/doi?doc=doi/10.1109/E-SCIENCE.2007.19, 2007.

[15] J.C. Preciado R. Morales-Chaparro, M. Linaje and F. Sánchez-Figueroa. MVC Web design patterns and Rich Internet Applications. http://www.sistedes.es/TJISBD/Vol-1/No-1/articles/SCHA-07-Morales-MVC.pdf, 2007.

[16] Georg Sander. Layout of Compound Directed Graphs, Technical Report A/03/96, University of the Saarlands, 5. Juni 1996.

[17] C. M. Sperberg-McQueen, Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and John Cowan. W3C Recommendation. http://www.w3.org/TR/2006/REC-xml11-20060816/, 16. August 2006.

[18] Achim Streit, Piotr Bala, and Alexander et al. UNICORE 6 – Recent and Future Advancements. http://hdl.handle.net/2128/3695, 2010.

[19] UNICORE Team. HiLA 2.1. http://www.unicore.eu/community/development/hila-reference.pdf, 2010.

[20] Michael zur Muehlen, Jeffrey V. Nickersona, and Keith D. Swensonb. Developing Web Services Choreography Standards – The Case of REST vs. SOAP. http://www.workflow-research.de/Publications/PDF/MIZU.JENI.KESW-DSS%282004%29.pdf, 2004.

Jül-4344
November 2011
ISSN 0944-2952